LEVERAGING THE USE OF API CALL TRACES FOR MOBILE SECURITY

Kobra Khanmohammadi

A Thesis

In the Department
of
Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of

Doctor of Philosophy (Electrical and Computer Engineering) at
Concordia University
Montreal, Quebec, Canada

February 2020

CONCORDIA UNIVERSITY

SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Kobra Khanmohammadi

Entitled: Leveraging the Use of API Call Traces for Mobile Security

and submitted in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY (Electrical and Computer Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ *Chair's name*	Chair
______________________________ *Examiner's name*	External Examiner
______________________________ *Examiner's name*	Examiner
______________________________ *Examiner's name*	Examiner
Abdelwahab Hamou-Lhadj	Thesis Supervisor
Raphaël Khoury	Thesis Supervisor

Approved by ___________________________________

Graduate Program Director

Month/day/year ______________________________

Dean's name

Faculty name

Faculty of Engineering and Computer Science

ABSTRACT

Leveraging the Use of API Call Traces for Mobile Security

Kobra Khanmohammadi, Ph.D.

Concordia University, 2020

The growing popularity of Android applications has generated increased concerns over the danger of piracy and the spread of malware. A popular way to distribute malware in the mobile world is through the repackaging of legitimate apps. This process consists of downloading, unpacking, manipulating, recompiling an application, and publishing it again in an app store. In this thesis, we conduct an empirical study of over 15,000 apps to gain insights into the factors that drive the spread of repackaged apps. We also examine the motivations of developers who publish repackaged apps and those of users who download them, as well as the factors that determine which apps are chosen for repackaging, and the ways in which the apps are modified during the repackaging process. We have also studied android applications structure to investigate the locations where malicious code are more probable to be embedded into legitimate applications. We observed that service components contain key characteristics that entice attackers to misuse them. Therefore, we have focus on studying the behavior of malicious and benign services. Whereas benign services tend to inform the user of the background operations, malicious services tend to do long running operations and have a loose connection with rest of the code. These findings lead us to propose an approach to detect malware by studying the services' behavior. To model the services' behavior, we used API calls as feature sets. We proposed a hybrid approach using static and dynamic analysis to extract the API calls through the service lifecycle. Finally, we used the list of API calls preponderantly present in both malware as well as benign services as the feature set. We applied machine learning algorithms to use the feature set to classify malicious services and benign services.

Acknowledgments

I am immensely grateful to and cannot thank my supervisor, Dr. Abdelwahab Hamou-Lhadj, enough for accepting to enroll me as a PhD student and for supporting me throughout this journey. Thank you for believing in my capabilities and skills and for guiding me to advance my knowledge.

I am very thankful to my co-supervisor, Dr. Raphaël Khoury, for his helpful advices and ceaseless support.

I am also very thankful to all my discerning committee members for their advice.

I am grateful to the Concordia School of Graduate Studies and the Natural Sciences and Engineering Research Council of Canada (NSERC) for their continuous financial support during my graduate studies.

Special thanks to Sheryl Tablan, the Graduate Program Coordinator - Ph.D. Program, who assisted me in navigating various administrative ambushes.

I want to thank my husband, Ali, and my daughter, Elena, for their welcoming distractions and support. Having them in my life always makes all my challenges easier to surpass.

Finally, I want to take this opportunity to remember my dearly beloved and much missed brother, Ebrahim. The one who had always been a hilarious and cool brother for me. All your lovely memories make me laugh again, while I am still grieving your loss. Rest in peace!

Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Problem and Motivations

The growing trend of smart phone usage has inspired the development of mobile applications (apps) to serve users in a variety of areas including entertainment, communication and more critical activities such as banking. The number of applications available for download in the leading app stores, Google Play Store or Apple's App Store, reached 2.1 million in March 2019 (Statistica 2019). The tremendous popularity of apps has set in motion the burgeoning development of malware. McAfee Labs Threat Report shows that as of August 2018, the number of the newly detected malwares exceeded 1,500,000 (McAfee 2018).

According to the Open Web Application Security Project (OWASP), application repackaging is one of the top ten mobile risks (OWASP 2016). Repackaging of a mobile application consists of downloading an app from an app store, decompiling it, changing its content, recompiling it, and finally uploading it to an app store again. This process is facilitated by the existence of open source tools such as APKtool (Wiśniewski 2012).

App repackaging has several negative effects. It threatens to create a loss of revenue for app developers through the republishing of paid original apps for no cost, or by removing the existing advertisements. To make matters worse, attackers often resort to the use of repackaging to spread malware. Zhou et al. showed that more than 85% of malware are introduced through app repackaging (Zhou and Jiang 2012b). Recent incident reports confirm that repackaging is still a popular way of distributing malware. For example, in April 2017, the malware "FalseGuide" was embedded in several repackaged apps, available on the Google Play Store (Kumar 2017).

The increased prevalence of malware is due mainly to the following reasons:

- Users are not aware that giving certain permission to an app might cause security issues.

- Attackers have the same capability to develop malware and upload them into marketplaces as legitimate developers.
- Although official Android app stores investigate apps before uploading them to the store it is not clear how these investigations are carried out and to what extent. The Arxan Technology (2015) report shows that some malware samples were uploaded in Google Play Store such as DroidDream Trojan in 2011.

The increasing number of malware samples found in Android apps has motivated researchers to develop several malware detection techniques. Studies conducted in detecting malwares in desktop applications are used to study the Android apps in order to identify the malicious behavior of the apps. Studying the similarity of apps is another detection approach, since malware uses repackaging to embed the malware in legitimate code (Zhou and Jiang 2012b). The challenges that malware detection techniques face can be summarized as follows.

- *Difficulties related to the use of machine learning algorithms*: Some approaches (e.g., (Alazab et al. 2010), (Islam and Altas 2012), (Sanz et al. 2012), (Wu et al. 2012), (Yang et. al 2014) and (Mariconti et al. 2017) (Chen et al. 2019)) use machine learning algorithms to learn the characteristics and behavior of malware and build clusters of malware families to detect zero-day exploits. Features used to learn malware behaviors are permissions (Barrera et al. 2010; Wu et al. 2012), intent (Yang et al. 2014), API (Alazab et al. 2010; Wu et al. 2012), system calls and smartphone features such as battery usage, memory, CPU and Network (Alam et al. 2013). The main drawback of these approaches is that several malware samples from the same family are needed to learn the behavior of malware in that family.
- *Dynamic loading and Native code*: Some malware like Base-Bridge and DroidKungFu Android malware (Zhou et al. 2012b) extract the actual malicious payload from external places rather than the original applications themselves. Thus, static analysis approaches (Wu et al. 2012) cannot detect them. It has been suggested that a study of OS interactions could allow the detection of malicious operations in native code (Tam et al. 2015). The process of detecting malicious code in dynamic loading and native code in Android apps still suffer from computational complexity.
- *Obfuscation*: Detecting malware based on signatures has always suffered from the problem of obfuscation. Obfuscation is the method of changing the code in a way that the sematic and

functionality of the code are maintained, but the code appearance is changed. To this end, studies (e.g., Burguera et al. 2011, Gascon et al. 2013, Luoshi et al. 2013, Suarez-Tangil et al. 2014, Yang et al. 2014 and Zhou et al 2012b) proposed using behavioral graph models to learn malware behavior. In these studies, the graph is used to find the structure of malware behavior, which is different from normal apps. These approaches suffer from computational and memory usage overhead. Moreover, they fail to identify certain usages of instances/methods, which are encrypted or use Java reflection and native code.

- *Curse of studying the similarity of Apps*: Some studies (e.g., Zhang et al. 2014, Zhou et al. 2012a, Shao et al. 2014 and Jiao 2015) focus on detecting the similarity between the repackaged app and the original version. Since the two apps (the original and the repackaged one) are designed to offer the same experience to the users, the repackaged apps will have a high chance of being downloaded because of having the same UI as the original version of the app. For example, Shao et al. (2014) studied the similarity of resources, such as images, in the apps. Besides the computational overhead of these approaches, the main constraint is that they need to compare apps two by two to find the similar ones. They also cannot detect if the repackaged version contains malware or just minor changes such as embedded ads.

The common practice in detection is to extract attributes from an app such as opcodes, API calls, images, resources, and user interface graphs and use them to detect the corresponding repackaged apps. Among these features, Android API calls invoked by the application components seem to be the most reliable (Au et al. 2012), given that it is difficult for an attacker to manipulate API calls.

Generating API call traces from an Android application is a challenging problem. In a normal situation, a simple static analysis technique would suffice. However, static analysis is limited when apps are obfuscated, i.e., their code is manipulated in a way that makes it hard to read and understand. One common obfuscation technique is based on the use of reflection, which consists of the ability for a class or object to examine itself at run-time. Using reflection, one can access the class attributes, dynamically invoke its methods, etc. (Forman et al. 2004). Reflection, a widely used obfuscation technique in repackaged apps, has often been discussed in related literature as the main obstacle for relying fully on static analysis for the detection of repackaged apps ((Hanna et al. 2013), (Guan et al. 2016) and (Hammad et al. 2018)). On the other hand, approaches based on dynamic analysis (e.g., Aldini et al. 2015 and Wu et al. 2015) suffer from a lack of code coverage because they run applications by providing incomplete data inputs. In addition, there exist malware families that hinder dynamic analysis by using techniques such as emulator detection techniques or adding logical bombs, specific time conditions that must be met before running the malicious operations. Limited work has been carried out using a hybrid approach, combining the advantages of both static and dynamic approaches. Hybrid approaches were used for some specific goals such as identifying reflection calls' parameters (Rasthofer et al. 2015), or for generating input data for malware analysis, as done by Wong and Lie (2016).

Despite the advances in the field, repackaging remains a serious threat, partly because it is still not well understood. In this thesis, we perform an empirical study to understand the factors that make apps vulnerable for being repackaged. We then focus on studying the behaviour of service components of Android apps as we show that malware distributors often resort to hiding the malicious code in these service components of repackaged apps. We also extract API calls in service components using a hybrid approach (combining static and dynamic analysis). To our knowledge, this is the first time that a hybrid approach is used to extract API calls from obfuscated apps. Finally, we leverage machine learning methods to detect malicious services using the extracted API calls.

1.2 Research Contributions

In this thesis, we make the following contributions (Figure 1.1):

- Empirical study of repackaged apps: We perform an empirical study in order to understand the factors that make apps vulnerable to being repackaged. We achieve this by empirically examining 15,296 pairs of original and repackaged Android apps, published in AndroZoo (Li et al. 2017a), one of the largest collection of Android apps used in research studies on mobile security. In this study, we address five research questions (RQ) as follows:
    - RQ1. What is the unfavorable prevailing usage of repackaging?
    - RQ2. How is the code of original apps manipulated to embed adware?
    - RQ3. Which type of apps have been exploited for repackaging?
    - RQ4.Why do users download the repackaged apps when the original versions are available for free?
    - RQ5. How are an app's attributes modified in the repackaged version?

- Exploratory study of service behaviors: We study the code of malicious services found in AndroZoo (Li et al. 2017) and Genome datasets (Zhou and Jiang 2012b). We examine the dependencies that the malicious services have with rest of the code and APIs called by services.
- API call trace extraction: We propose a hybrid approach combining static and dynamic analysis to extract API calls invoked in the service components of Android apps.
- Detecting malware by behavior analysis of a service component through the API call traces: To differentiate a benign app from the malicious one, we use the proposed hybrid approach to extract trace of API calls through a service lifecycle. We use machine learning algorithms to model service behavior and detect malicious services.

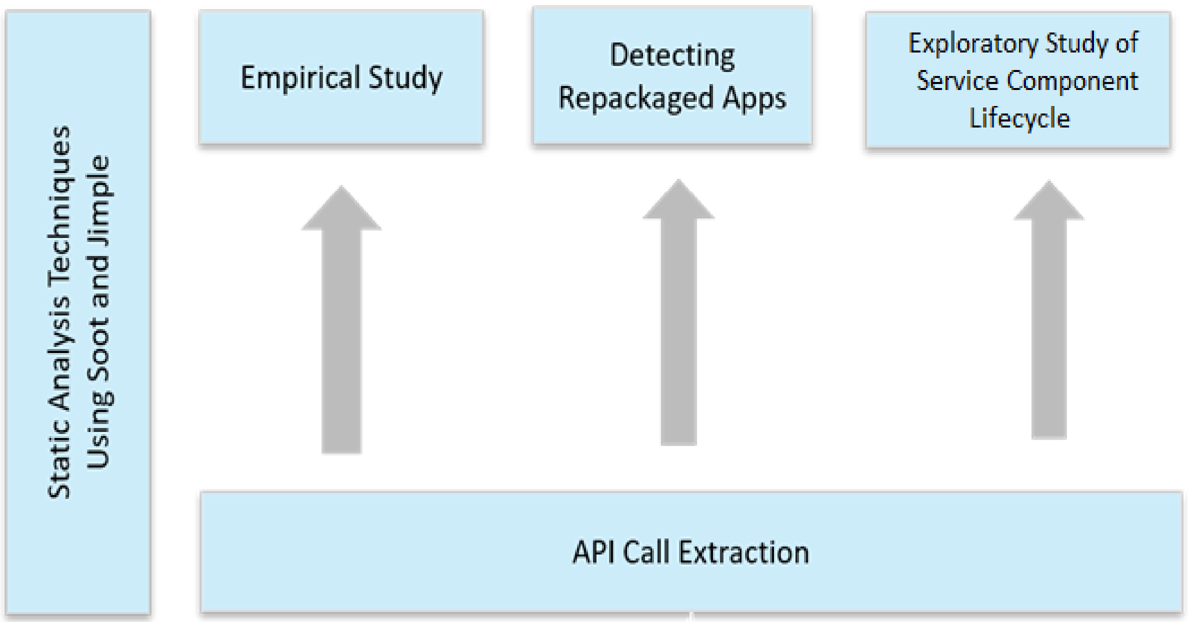

Static Analysis Techniques Using Soot and Jimple
Empirical Study
Detecting Repackaged Apps
Exploratory Study of Service Component Lifecycle
API Call Extraction

Figure 1.1 Overview of the thesis contributions

1.3 Thesis Organization

The thesis organization is as follows: in Chapter 2, we provide background information about Android app and their service components. In Chapter 3, we present and discuss related studies. Chapters 4, 5, 6, 7 and 8 are dedicated to the main contributions of this thesis. Finally, we conclude the thesis in Chapter 9, and discuss the limitations and future directions.

1.4 Related Publications

Earlier versions of the work presented in this thesis have been published in the following papers:

1- Khanmohammadi K, Rejali M, Hamou-Lhadj A (2015) Understanding the Service Life Cycle of Android Apps: An Exploratory Study. In Proc. of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), Denver, US, pp 81-86 (Contribution 2)

2- Khanmohammadi K, Hamou-Lhadj A (2017) HyDroid: A Hybrid Approach for Generating API Call Traces from Obfuscated Android Applications for Mobile Security. In Proc. of the 2017 IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, Czech Republic, pp 168-175 (Contribution 3)

3- Khanmohammadi K, Ebrahimi N, Hamou-Lhadj A, Khoury R (2019) Empirical Study of Android Repackaged Applications. Empirical Software Engineering, Springer, 24(6): pp 3587-3629 (Contribution 1)

4- Khanmohammadi K, Hamou-Lhadj A, Khoury R (2019) On the Use of API Calls to Detect Repackaged Malware Apps: Challenges and Ideas. In Proc. of the 30th International Symposium on Software Reliability Engineering (ISSRE 2019). (Contribution 2)

Collaborations:

5. Ebrahimi N, Trabelsi A, Islam MS, Hamou-Lhadj A, Khanmohammadi K (2019) An HMM-based approach for automatic detection and classification of duplicate bug reports. Information and Software Technology, Volume 113, pp 98-109.

Chapter 2 Background

2.1 Android Architecture

The Android operating system is a stack of software components, roughly divided into five sections and four main layers, as shown in Figure 2.1, taken from the architecture diagram in Android Developer Documentation (2018).

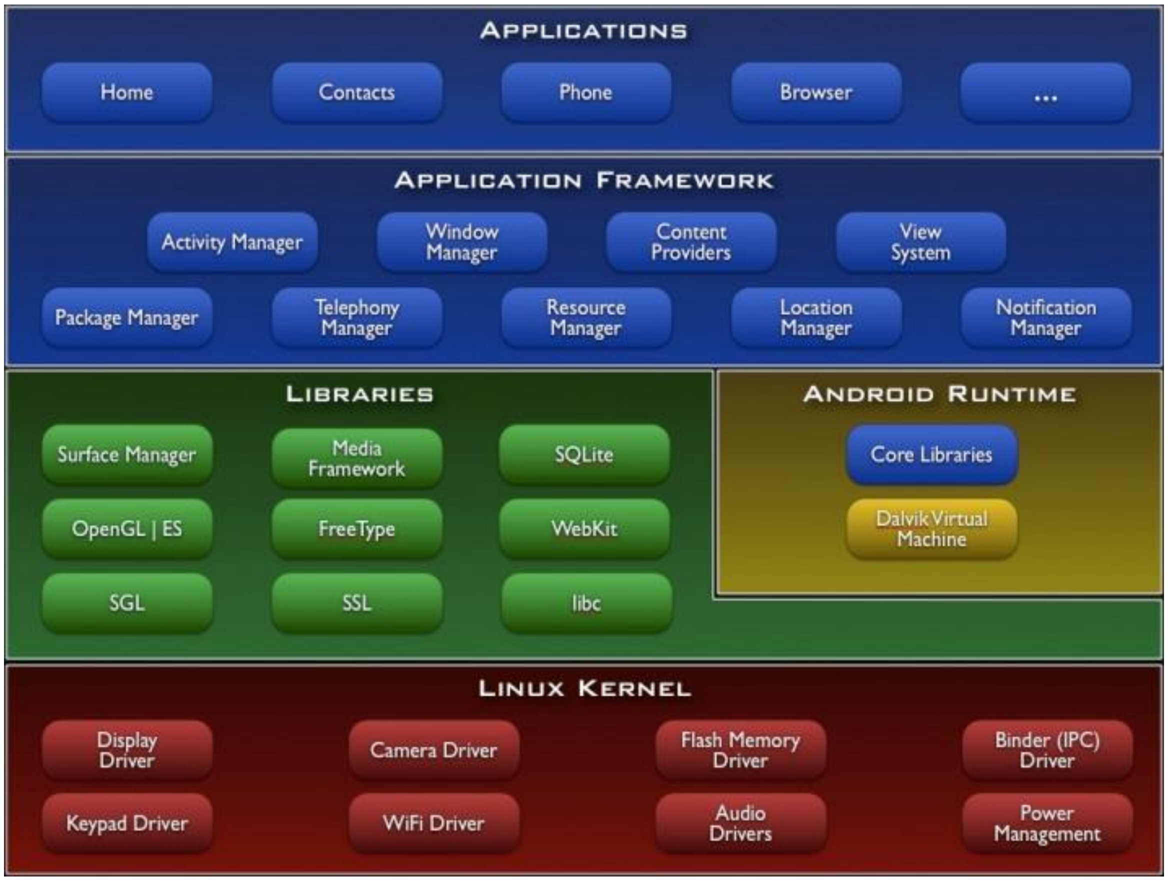


Figure 2.1 Android architecture (Android Developer Documentation 2018)

Each Android section is explained in more detail in what follows:

- Application Layer: User applications run in this layer. These applications are mostly written in Java, packaged with .apk suffix and will execute in Dalvik virtual machine.

- Application Framework: This provides several higher-level services to applications in the form of Java classes called Android APIs. Some key Android services provided by the application framework include Activity Manager, Content Providers, and Notifications.
- Android Runtime Section: This section provides a key component called Dalvik Virtual Machine, which is a kind of Java Virtual Machine specially designed and optimized for Android. The Dalvik VM makes use of Linux core features such as memory management and multi-threading and enables every Android application to run in its own process, with its own instance of the Dalvik virtual machine. This separation provides sandboxing for every application running in the device.
- Libraries: This section contains a set of libraries, including the open-source Web browser engine WebKit, the well-known library libc, and the SQLite database, which is a useful repository for storing and sharing application data. As another example, there are libraries to play and record audio and video. In addition, there are SSL libraries responsible for connecting to the Internet and Opengl library, which provides a Java interface to the OpenGL ES 3D graphics rendering API.
- Linux kernel: The kernel includes drivers for hardware, networking, file system access and inter-process-communication.

2.2 Android Application Components

An Android app is uploaded as a zip file with extension ".apk" to an app store. This file generally contains an app program in the form of a classes.dex file, as well as app resources such as pictures, music and .xml files, which describe the layout information. It is also required to contain the file AndroidManifest.xml, which contains information about the app- its name, version, access rights, app components and referenced libraries. The file Classes.dex is in the format used by the Dalvik Virtual Machine provided in Android.

When an app is about to be installed on a device, the user is prompted to grant it permissions to access device resources such as the network, disc storage, etc. Users can either accept the requested permissions or refuse the installation of the app. Moreover, for the apps that are built for Android 6.0 and up, you can allow or deny permissions once you start using them. The tag <*uses-permissions*> in the AndroidManfest.xml lists the permissions that are requested by the app. Note that starting from Android Marshmallow 6.0 (released in October 2015), apps can also ask for permissions at runtime.

Android apps are usually written in Java and contain four types of components: Activities, Services, Content Providers and Broadcast Receivers, which correspond to four main Java classes, namely Activity, Service, Receiver, Content Provider. Each component of a specific type must inherit from the corresponding superclass. Activities capture what a user can do with the app. They implement the user interface. Services run in the background and usually contain long running operations. Content providers manage access to a central repository of data among a device's apps. Broadcast receivers are used to respond to system-wide broadcast announcements such as *"the battery is low"*. More details about components are available on Android Developer Documentation (2018).

After installation, each application runs in a separate user space process as an instance of the Dalvik Virtual Machine (DVM) and usually with a distinct user and group ID. Although isolated within their own sandboxed environment, applications can interact with each other and with the system through well-defined APIs.

Message objects called "intent" are designed for activating activities, services and broadcast receivers. Components communicate with each other via Inter Process Communication (IPC) and intents are the primary vehicle for IPC. For example, an activity may send "intent" to display the user's current location on a map. To develop it, an intent containing the user's location will be defined and sent to a component that renders the map. The intent's target component can be defined in two ways, (a) explicitly, by specifying the target's application package and class name, and (b) implicitly by setting the intent's action, category or data fields. In order for a component to be able to receive implicit intents, "intent filters" have to be specified for it in the application's manifest file. IPC can occur both within a single application or between different applications.

2.3 Service Component of an Android Application

A service is a component that runs in the background to perform long-running operations or to perform work for remote processes. A service does not provide a user interface. For example, a service might play music in the background while the user is in a different app, or it might fetch data over the network without blocking user interaction with an activity. In AndroidManifest.xml, a service is defined by a <*service*> tag. It has attributes such as *name*, *exported* attributes that shows if other applications can use this service, and an *intent filter*, which identifies the type of intents that the service accepts. A service will be implemented in Java as a subclass of class *Service*. In the *AndroidManifest.xml*, the attribute *android:name* in the <service> tag specifies the class name of the service component in the corresponding Java code.

There are three ways to launch a service in an Android application, “started” , “bound” or “Scheduled”. In the first case, a service is started by an another component of the app by calling the method startService(). Once it is started, the Android system will run the service indefinitely in the background even if the user switches to another application. It will be stopped by calling either the stopSelf() or the stopService() method. In the second case, a component is bound to a service by calling the method bindService(). A bound service offers a client-server interface, allowing that component to interact with. It is possible to bind more that one component to a single service simultaneously. A component unbinds from a service by calling the method unbindService(). A bound service is destroyed by the Android system as soon as all of its client components are unbound. Usually, for long running operations and single tasks, a started service will be used. In the third case, a service is launched at a scheduled time by using an API call such as the JobScheduler, introduced in Android 5.0 (API level 21).

The *Service* class provides a number of methods, called lifecycle callback methods, that allow the service to know that a state has changed: that the system is creating, stopping, or resuming a service, or destroying the process in which the service resides. The Android system will call callback methods often in a specific order while a service is launched. For example, when a service is started by another component (such as an activity) by calling startService(), a typical sequence of callback methods could begin by calling onCreate(), then followed by calling onStartCommand() and finally it may finish by calling onDestroy(). Developers should implement their operations in callback methods to let the Android system execute them while the service is launched. Therefore, we need to go through each of these callback methods to extract the API call traces through the service lifecycle.

The following figure shows a class diagram of a malicious service in malware called “GoldDream” (Symantec Report 2011a). As it is shown, in class “zjService”, there are callback methods and some other methods that, obviously, should be called in the callback methods if it needs to be executed in the service lifecycle.

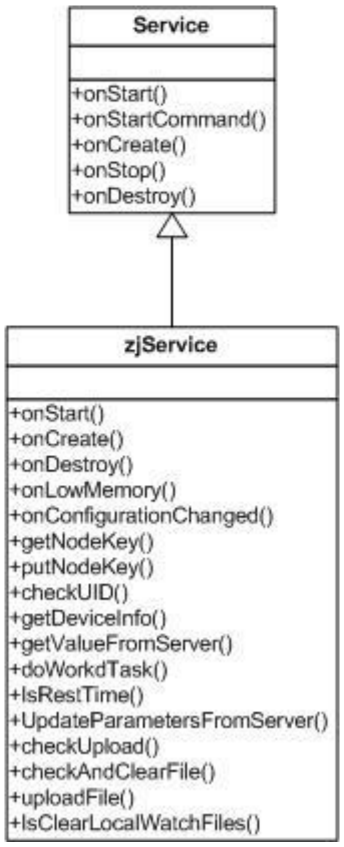


Figure 2.2. Class diagram of service zjService in malware GoldDream

Chapter 3 Related Work

3.1 Introduction

In this chapter, we present studies related to the topic of this thesis with a focus on the detection of repackaged apps, regardless of whether the repackaged app is malware or not as well as studies that tackle the problem of detecting malware in Android apps in general.

3.2 Detecting Repackaged Applications

3.2.1 Code Similarity

Early work on detecting repackaged apps relied upon pair-wise comparisons between two apps to see if they showed similar behavior and attributes. It assumes that the repackaged app's appearance is similar to the original app. Researchers proposed using a variety of factors to identity repackaged apps including: instruction sequences (Zhou et al. 2012a), static data dependency (Crussell et al. 2012) (Crussell et al. 2015), method graph (Potharaju et al. 2012), and k-grams of binary opcode sequences (Hanna et al. 2012).

Zhou et al. (2012a) measured the similarity of apps based on similar instruction sequences on the code of apps. They leveraged a specialized hashing technique, called fuzzy hashing. In this approach, a hash value is computed for each local unit of opcode sequence of the *classes.dex*. To localize the modification caused by repackaging, it uses a reset point to split long opcode sequences into small units and then concatenates all of the hash values into a whole. They tested their approach by computing the similarity between apps in the official Android marketplace and in third-party market places. The drawback of this approach is that apart from the computational overhead, obfuscation techniques may also alter the code. They also could not detect whether the repackaged version contained malware or if it just had minor changes such as the addition of advertisements.

Crussell et al. (2012) proposed an approach, called DNADroid, to detect similar apps. It computes the static data dependency graph (DDG) of every method and two apps sharing similar DDG are identified as similar apps. Later, Crussell et al. (2015) proposed a tool, called AnDarwin, which decreases the computation time by eliminating pair-wise comparison. AnDarwin creates clusters of features based on their app sets; and relies upon it to detect partial app similarity by finding similar sets. It also removed external libraries to increase the accuracy.

Hanna et al. (2013), with the goal of detecting buggy and vulnerable code reuse (codes downloaded from vulnerable libraries), known malware instance and pirated apps, studied the Dalvik code of apps. They represented each application by the hash of the extracted features. The features are k-grams of Dalvik opcode sequences.

Potharaju et al. (2012) also present an approach to find similar apps based on the app's method similarity. They recorded Abstract Syntax Tree which contains methods' information such as the number of arguments and the methods invoked by each method. Androguard (Desnos 2015) also tried to find the similarity of two android applications by checking the similarity of methods of the apps. Androguard is a free static analysis tool applied over the Smali intermediate code of an android app. It uses histograms and classic Shannon Entropy (Shannon and Weaver 1948) to determine the probable similar methods when their similarity is above a threshold.

Guan et al. (2016) presented an approach called RepDetector. It performs static analysis to extract lists of classes, functions and their parameters. It uses input-output states of core functions in the app and then compares function and app similarity.

All of these approaches require performing a static analysis of the app code and suffer from obfuscation techniques such as reflection and encryption. To combat the limitations of static analysis, dynamic analysis approaches have been proposed. Wu et al. (2015) modeled the app's behavior from the HTTP traffic. Aldini et al. (2015) detect similar apps by processing the system call traces logged through the execution of an app.

3.2.2 User Interface Similarity

Some researchers have proposed detection techniques that rely upon a visual inspection of the appearance of the apps. Their work is based on the fact that manipulation of apps in repackaging is done in such a way as to maintain the same sense of "see" and "feel" of the original app in order to encourage users to download the repackaged apps.

As mentioned in Section 2, activities are the components in Android apps that implement the user interface. Zhang et al. (2014) present an approach, called ViewDroid, to detect similar apps. They construct a graph of activity classes in an app by using the tool Androguard (Desnos 2015). Each activity is a node in the graph and is defined by Android APIs invocation; the graph edges are based on the methods, such as *onclick()*, triggered in calling other activities. They compute the similarity between two apps based on isomorphic graphs in two apps. To evaluate their approach, they compared in pairs all apps which are in the same category in app stores. Their comparison approach, obviously, had a high computational cost.

Jiao et al. (2015) used an image processing approach, called pHash, to find the similarity between images in two applications to detect repackaging. They utilized a new storage method in the form of {features, application} which lead to detecting apps with the same features faster. In fact, apps with the same features are stored in the same storage. Therefore, it decreases the number of comparisons needed.

Sun et al. (2015) proposed an approach to check UI layout similarity by studying the xml files in the layout folder of the apps presenting information about UI. Soh et al. (2015) detect Android app clones based on the analysis of UI information collected at runtime. Chen at al. (2015a) detect repackaged apps by detecting similar UI structure within apps. Zhauniarovich et al. (2014) proposed an approach to compare the resource in the app package (.apk file). Kywe et al. (2014) proposed detecting similar apps by applying text similarity and image similarity measurements. Yue at al. (2016) introduced RepDroid, a tool that relies upon traces of UI to detect repackaging. Their method benefits from the fact that it does not necessitate the apps' source code, and is thus resilient to obfuscation. Nguyen et al (2019) mapped the map of code's hash with UI images as a signature of an app and proposed to detect the repackaged apps when two apps have similar maps.

With the same assumption that a repackaged application has the same "see" and "feel", Shao et al. (2014) extracted five statistical features including number of activities, number of permissions, number of intent filters, number of .png files per drawable directory and average number of .xml files per directory res. (resource directory in an apk file), as well as the average number of references to the 10 most-referenced resources such as string, color, and style. These statistics gave information about the resources of the app. They used the nearest neighbour search and clustering to classify the apps into small groups. The approach needs another step of manual investigation to determine if the apps in each group are repackaged.

These approaches also suffer from the extensive computation time of pair-wise comparisons. Some researchers proposed approaches that require less computation time. For example, Jiao et al. (2015) used an image processing approach to find the similarity between the images present in two apps. They employed a new storage method in the form of features-application pairs, which stores apps with the same features in the same data structure. This approach decreases the number of comparisons needed. Zhou et al. (2013b) proposed an alternative two-step method, based upon a static analysis of the code. First, they develop a module decoupling technique that can separate the main functionalities of the program from third party libraries. Second, they extract a vector of semantic features from the module implementing the main functionalities of the program, which allows rapid detection of similar code.

3.2.3 Reverse Engineering Symptom

Gonzalez et al. (2014) proposed the idea of detecting repackaged apps based on the telltale signs left by the tools that perform reverse engineering. An Android app contains an ordered list of the string identified in the app's source code. The ordering of the strings in this list changes after repackaging, yielding evidence of repackaging.

3.3 Malware Detection

Studying the similarity of apps improves malware detection since some malware codes are embedded in legitimate apps through repackaging (Zhou and Jiang 2012b). There is no perfect method that can provide the desired accuracy in detecting malware in repackaged applications. In this regard, we summarize the related work based on their analytical approach: 1- static analysis, 2- dynamic analysis and 3- hybrid approaches. To present all of the drawbacks in this area, we added another category for approaches that used machine learning in their analysis. Note that, this category includes literature from the other three categories.

3.3.1 Static Analysis

Host-based detection- Some research studies (e.g., (Arp et al. 2014), (Arora et al. 2014), (Saracino et al. 2016), and (Sun et al 2017)) focus on host-based online analysis after an application has been installed. They monitor its activities and study the recorded operations to detect malicious ones. Among them, Arp et al. (2014) proposed a tool, called Drebin, which utilizes static analysis to identify malicious applications. They implemented a lightweight disassembler to extract features: restricted API calls (which needs permissions), used permissions, suspicious API calls (API calls that allow access to sensitive data or resources), and network addresses. The extracted features are stored in a joint vector space and used by the Support Vector Machine (SVM) algorithm for the classification between malware and normal applications. Due to smartphone's limitation in power, memory and CPU, it is preferable to carry out the malware detection offline in the app store before installing them. The studies that proposed approaches for offline detection of malware in app stores are summarised as follows.

Signature based detection- Some papers proposed static analysis to study the code of apps and uncover the signature of malware ((Luoshi et al. 2013), (Fan et al. 2015), (Shahriar and Clincy 2014) (Hu et al. (2014), and (Zhou et al. 2012b)). For example, Shahriar and Clincy (2014) used the Kullback-Leibler Distance (KLD) to identify altered legitimate applications. They computed the probability of occurrence of specific opcodes that may indicate likely malicious operations, such as a message sending operation. When the KLD of the probability for these opcodes in two apps is less than a threshold, the apps are similar to each other and are may be repackaged. Suarez-Tangil et al. (2014) extracted the code chunks of the app's code through static analysis and used them as a feature to classify malware samples (Note that a code chunk is a piece of code without a branch). They detected the apps containing malware by comparing code chunks with the classified samples of malware.

Studies that propose signature-based detection of malware look for specific patterns in application codes. They can be easily by-passed by bytecode-level transformation attacks (Rastogi et al. 2013). Two groups of research works have sought to address this problem. The first group tried to identify the semantics of known risky and malicious behaviors rather than a specific signature. The second group applied machine learning algorithms to learn malware behaviors and detect anomalies.

Semantic based Detection- To learn the semantic of malware, some researchers used API calls for studying the behaviour of malware since it is hard to replace Android APIs with the new ones while preserving the functionality. Luoshi et al. (2013) extract some sensitive APIs called mostly in malware samples such as *TelephonyManager.getDeviceID()*. Gascon et al. (2013) used static analysis to construct the method call graph for samples of malware in the same malware family. Detection of malware is based on comparing method graphs of applications and malware families.

Hu et al. (2014) proposed MIGDroid which extracts method graphs from the smali[1] representation of bytecode of apps and scores each method based on the sensitive API invocation. Methods with high scores and with less connectivity with the rest of the code are detected as potential malware codes.

[1] https://github.com/JesusFreke/smali/wiki

Grace et al. (2012) proposed the tool, called Riskranker, to analyze whether a particular app exhibits dangerous behavior (e.g., launching a root exploit or sending background SMS messages). The output is a list of potential malware apps that merit further investigation. They were successful in detecting zero day malware, as they study existence of the critical operations in apps instead of signature of malware.

Machine learning approaches- Some studies use machine leaning algorithms over the extracted features from apps' code to detect malware. Yerima et al. (2013) used Baysian Classification over the features, including API calls, Linux system commands, permissions, the presence of encrypted code and secondary .apk or .jar files. Yang et al. (2014), in DroidMiner, also extracted the call graphs of applications and the vector of sequence API calls in methods. They used them as the features in machine learning algorithms for classifying malware and benign applications. Aafer et al. (2013), in DroidAPIMiner, used frequent API calls in applications as the feature for classifying malware and benign applications. They only used API calls, which are highly frequently found in malware. For those APIs, they performed data flow analysis to recover their parameter values and selected only the APIs that invoke dangerous values. They also removed the API calls that were exclusively invoked by third-party packages such as advertisement packages. They applied three different classification algorithms: Decision Tree (Quinlan 1986), Nearest Neighbor (KNN) (Fix et al. 1951), and linear Support Vector Machines (SVM) (Schölkopf and Smola 2002). In the most recent research, Mariconti et al. (2017) proposed MaMaDroid. It uses the API sequences as the training feature for classification. It extracts the call graph of API calls in an application and builds the Markov chains over API calls in each code path as the feature vector for classification. They evaluated classification system by performing a variety of algorithms including Random Forests (Breiman 2001), 1-Nearest Neighbor (1-NN), 3-Nearest Neighbor (3-NN), and Support Vector Machines (SVM). They compared MaMaDroid against DroidAPIMiner and showed that it outperforms DroidAPIMiner.

Drawback of using static analysis- The main problem with static analysis is obfuscation approaches that may completely preclude detecting repackaged applications as well as malware. In obfuscation, the code is changed in a way that preserves the sematic and functionality of the code, but alters the appearance of the code. All of the studies that use behavioral graph models to learn malware behavior (e.g., (Burguera et al. 2011), (Suarez-Tangil et al. 2014), (Luoshi et al. 2013) (Gascon et al 2013), (Yang et al. 2014) and (Zhou et al. 2012b)) failed to identify certain usages of instances/methods, which contain encryption or use java reflection or native code. For example, Gascon et al. (2013) evaluate their approach against some obfuscation techniques. They showed that their approach is resistant to some obfuscation techniques, such as changing the name of methods. However, the call graph could still be obfuscated by the addition of unreachable calls. Moreover, function inlining can be used to hide the graph structure.

Another issue in doing static analysis in android applications relates to the difficulty of finding data flow. Because of Inter Process Communication (IPC) in android apps, finding the control flow is not similar to previous approaches working in java codes. Finding the data flow and communication between components through static analysis requires studying implicit calls of target components in IPC. Octeau et al. (2013) provided a tool, EPIC, which identifies components' relations. This tool presented the graph with all possible connections between components that are determined by studying the *intent filters* in *AndroidManifest.xml* of the apps and studying the code of components to find places where new components were created. Their approach relied on the success of analysing the source code.

3.3.2 Dynamic analysis

Zhou et al. (2012b) showed that some samples of malware families like *Basebridge* and *DroidKungFu* malware extract the actual malicious payload from external places rather than the original applications themselves. Thus, static analysis approaches are not able to detect them. Researchers use dynamic analysis to detect the malicious operations implemented in the code loading, to study the behavior of native code and to solve the problem of code obfuscation techniques such as reflection since the reflected method will be called at run time.

Lin et al. (2013) study the signature of malware families based on the system call recorded while the app is running. Enck et al. (2014) also proposed a dynamic analysis tool called TaintDroid. It tracks data from sensitive sources to sensitive destinations and report to the users how their sensitive data are being used by various apps. Tam et al. (2015) also used dynamic analysis to study system calls and their parameters. In their framework, presented earlier by Reina et al. (2013), a modified Android emulator called CopperDroid, is run to collect the invoked system calls. To understand the high level behavior of the app, CopperDroid parsed the payload of the ioctl system call (input/output control system call), and sent the extracted arguments to the un-marshalling part of the framework in an unmodified android to reconstruct the called API and its parameters. They used this framework to reconstruct the behavior of malware. They triggered and disclosed additional behavior in more than 60% of analyzed malware samples. In fact, this dynamic analysis solved the problem of obfuscation technics such as reflection and encryption. Even though this dynamic analysis gives better behavior representation for malware's malicious behavior, detecting the part of the code in normal application still needs a powerful imitator for user interaction in order to trigger the part of the code that has malware. In cases where there is a logic/time bomb or emulator detector in the code, it may not execute the malicious part of the code.

Some malware samples use emulator detection system to evade the running of code on the emulator. Petsas et al. (2014) presented how a malware can use heuristic data obtained from the system it is running on to detect the emulator. They have presented the taxonomy of these heuristic data by studying malwares that evade dynamic analysis with the use of emulator detection. They also suggested some heuristics and tested them on the existing emulators. They showed that there does not exist single emulator that can defeat every type of emulator detection in existing malware. Working on the perfect emulator, similar to a real android system, is still an open issue.

To avoid the problem of emulator detection in malware samples, Burguera et al. (2011) presented an approach, Crowdroid, which obtains traces from real devices. They implement a client/ server framework that allows client apps to send system call traces to a central server. This tool collects different samples of application execution traces, which are then used to differentiate the benign applications from those containing malware.

3.3.3 Hybrid analysis

Hybrid approaches, using both static and dynamic analysis, have also been proposed in order to benefit from both approaches' advantages in detecting malware. Rasthofer et al. (2015) presented a tool, called Harvester, in which static analysis extracts parts of the code containing reflective calls. They also instrumented the conditions for obtain high code coverage. Then, they used dynamic analysis to run that part of the code to identify the method name called by reflection. Finally, they changed the reflected calls to direct calls. Their approach in combination with a static analysis tool, FlowDroid (Arzt et al. 2014), is used for detecting malware. Wong et al. (2016) presented a hybrid analysis approach, called IntelliDroid, for malware detection. First, they located the sensitive APIs and studied the conditions in the applications' code, which could lead to them, such as the APIs provided for sending messages. Based on these conditions, they identified the inputs that are passed to these specific methods. Finally, in the dynamic analysis phase, they ran the code with the identified inputs. Their approach covers code paths that contain sensitive API calls. However, their approach did not handle obfuscation.

Sounthiraraj et al. (2014) proposed an approach called SMV-Hunter; it first uses static analysis to identify suspicious vulnerable applications and then applies dynamic analysis to detect the potentially vulnerable code by testing an active Man-in-the-Middle attack on them. Zheng et al. (2012) proposed an approach, called Smartdroid, to identify the paths from UI-based conditions into the sensitive behavior in static analysis phase. They applied dynamic analysis to go through the paths and expose the malicious operations. Yang et al. (2013) proposed hybrid analysis, called Appintent, to determine data transmission, which is not triggered by the application's user. It uses static analysis to identify a sequence of GUI events that could lead to data transmission. Again, here dynamic analysis is used to examine the potential event sequence cause the data transmission. AppAudit proposed by Xia et al. (2015) and ContentScop proposed by Zhou and Jiang (2013) used a combination of static and dynamic analysis approaches for detecting a particular kind of malware. Their approach relies mainly on static analysis to detect applications susceptible to special type of vulnerabilities. These techniques also focus on specific attacks. We will propose a generic hybrid approach, HyDroid, for detecting any attacks that manifest themselves through the application's services.

3.3.4 Difficulties related to the use of machine learning algorithms

Some approaches (e.g., (Burguera et al. 2011), (Alazab et al. 2010), (Islam and Altas 2012), (Sanz et al. 2012), (Wu et al. 2012), (Yang et al. 2014), and (Alam and Vuong 2013)) use machine learning algorithms to learn the characteristics and behavior of malware and build clusters of malware families to detect zero-day malware. Several features are used to learn malware behaviors including permissions ((Alazab et al. 2010), (Sanz et al. 2012), (Wu et al. 2012)), intent (Yang et al. 2014), API calls ((Alazab et al. 2010), (Islam and Altas 2012), (Wu et al. 2012)), system call traces (Burguera et al. 2011), strings contained in the application (Sanz et al. 2012), and smartphone features such as battery usage, memory, CPU and Network (Alam and Vuong 2013). Prominent machine learning algorithms such as Bayesian network, decision tree (random forest and j48), k-nearest neighbour, and support vector machines, are used to classify the apps with these features. Extraction of features in these approaches is done by both static analysis ((Alazab et al. 2010) and (Sanz et al. 2012)) and dynamic analysis ((Burguera et al. 2011), (Islam and Altas 2012), (Wu et al. 2012), (Yang et al. 2014), and (Alam and Vuong 2013)). The main drawback of these approaches is that they require multiple malware sample in a family to learn their behavior.

Chapter 4 Empirical study of repackaged apps

4.1 Introduction

According to the Open Web Application Security Project (OWASP), application repackaging is one of the top ten mobile risks (OWASP 2016). Several studies (e.g., Zhang et al. 2014, Shao et al. 2014 and Jiao et al. 2015) sought to automatically detect repackaged (and malicious) apps. The common practice is to extract attributes from an app such as opcodes, method calls, images, resources, and user interface graphs and use them to detect the corresponding repackaged apps. Despite the advances in the field, repackaging remains a serious threat, partly because it is still not well understood.

In this chapter, we perform an empirical study in order to understand the factors that make apps vulnerable to being repackaged. We achieve this by empirically examining 15,296 pairs of original and repackaged Android apps, published in AndroZoo (Li et al. 2017a), one of the largest collection of Android apps used in research studies on mobile security.

This study addresses five research questions:

RQ1) What is the unfavorable prevailing usage of repackaging?

Repackaging is used for different purposes such as revenue manipulation through advertisement, piracy, and the introduction of malware. After examining the repackaged apps, we found that repackaging is mainly used to introduce adware as opposed to other types of malware such as Trojans, etc. Adware is defined by Erturk (2012) as any software package that automatically presents advertisements to users by guessing from their previous surfing or search activities. Symantec (Chien 2005) additionally stresses that adware might perform other malicious activates, such as redirect a user’s searches to advertising websites, and collect personal data about users. Gao et al. (2019) likewise classify adware as a type of malware, and found that adware commonly performs multiple malicious operation, alongside with displaying ads. Adware is commonly classified as a subclass of malware (e.g. (Gupta 2013) and (Xue et. al 2017)) with distinctive objectives and modus operanti. We conclude that the main motivation behind app repackaging is to gain revenues through advertising.

RQ2) How is the code of original apps manipulated to embed adware?

Drawing upon our findings from RQ1, we focus on adware, and study how its introduction in apps leads to modifications in the apps' code. More particularly, we examined the API calls present in the parts of the code manipulated in repackaged apps and compared it to API calls present in the original apps. We found that the API calls that are added to repackaged apps are usually the same ones that are already present in the original apps. As a consequence, malware detection based on an analysis of API calls alone may be impractical.

RQ3) Which type of apps have been exploited for repackaging?

To answer this question, we investigate the relationship between the popularity of an app and its likelihood of being repackaged. To this end, we propose three metrics to quantify the popularity of an app: user rating, the number of downloads, and the popularity of the app store from which the app can be downloaded. We observed that apps that have a rating greater than 3 out of 5 are more likely to be repackaged. Moreover, most repackaged apps are modifications of apps downloaded from the Google Play Store. In addition, we studied the use of obfuscation and applied static analysis to identify the obfuscations techniques used in the original apps. Obfuscation techniques include method name changes, reflection and dynamic loading. Our findings show that most of the apps that get repackaged do not use name changes. This result suggests that the ease of understanding the code of an app is a major factor that leads the app to be targeted for repackaging.

RQ4) Why do users download the repackaged apps when the original versions are available for free?

To answer this question, we examined the popularity of repackaged apps considering their star rating, download number and the app store, which they are published. We found that many repackaged apps exhibit a high user star rating, high download number and were published in popular app store, which shows that they were successful in seducing users to download them. In addition, we examined the names of the repackaged apps to understand how they differ from those of the original apps. We found that repackaged apps usually have high star–rating, just like the original ones. In addition, the names of repackaged apps are usually kept similar to the original names, possibly to increase the odds of it being found by search engines. One interesting observation is that when the names of the repackaged app are changed, the new name is often in a different language than the original application, perhaps to target users from a specific region.

RQ5) How are an app's attributes modified in the repackaged version?

We studied how a selected set of apps' attributes are altered after the apps are repackaged. We observed that the size of apps may or may not increase after repackaging. We analyzed the number of app components and their names in the original apps and their repackaged counterparts. We found that the number and the names of components in repackaged apps are similar to those of the original versions. Another attribute that we studied is an app's permissions, which are used to access the mobile device's resources. We found that the number of permissions requested by an app generally remains the same after it is repackaged. Based on these findings, we propose an indexing scheme to record the apps in order to decrease number of comparisons needed to detect repackaged apps. The results of this chapter is published in a paper (Khanmohammadi et al. 2019a).

The remainder of this chapter is organized as follows: In subsection 4.2, we present related work on empirical study of Android apps. In subsection 4.3, we present the study's methodology. Then, we present each research question in a subsection and answer them based on findings. In subsection 4.9, we propose an indexing schema to record apps based on the finding in our study. In subsection 4.10, we discuss the threats to the validity. Discussion and concluding remarks are given in Section 4.11.

4.2 Related Work

Empirical study. A group of studies focuses on empirical studies of apps downloaded form markets to glean insights on the state of the art in repackaging. Mojica et al. (2012) put forth an empirical study on software reuse over apps downloaded from variety of categories in Google Play Store. They studied and identified similar classes in apps. Their findings show that app developers perform substantial software reuse. Mojica et al. (2014) also performed studies on class reuse in apps. They showed the percentage classes in apps inherited from a base class in the Android API, inherited from domain-specific base class, and computed the percentage of classes reused in each category.

Linares-Vásquez et al. (2014) studied the impact of third-party libraries and code obfuscation practices on estimating the amount of reuse by class cloning in Android apps. They showed that by excluding third-party libraries from the analysis, the amount of class cloning significantly decreased. They also found that obfuscation increases the number of false positives when detecting class clones. Viennot et al. (2014) proposed an approach for crawling the Google Play Store and downloading a large number of apps. In a recent study, Li et al. (2017) gathered large number of apps and proposed a code similarity metric in order to identify repackaged apps and provided a dataset of original and repackaged apps. They also carried out an empirical study over the dataset and provided findings to answer some research questions. More particularly, they have provided information on the way the piggybacked apps are changed. In comparison to their work, we aimed to answer research questions by considering a set of findings related to such research questions.

Adware detection. In this chapter, we will show that large numbers of repackaged apps are adware. Several studies have proposed approaches for detecting third-party libraries, including advertisement libraries, embedded in apps (e.g. (Ma et al. 2016) and (Backes et al. 2016)). Often, the motivation for this detection effort is that apps with embedded third-party libraries might also contain embedded malicious code. However, none of the studies we surveyed specifically detect malicious ad libraries versus benign ones.

Some studies have focused specifically on detecting malicious operations in Android adware. Liu et al. (2014) proposed a system called DECAF for detecting the placement of fraud of advertisements in apps. DECAF detects if an ad is shown in a manner that contravenes the relevant policies provided by ad networks such as AdMob (2019) or Microsoft Advertising (2019). For example, such policies may forbid an ad from being displayed too close from an app's UI button, in order to avoid a situation where the user clicks it unintentionally. Another fraud in adware is called on-click fraud. It occurs when adware fetches ads without displaying them to the user, or "clicks" on ads automatically. Crussel et al. (2014) focus on detecting on-click fraud by analyzing network traces. In a recent study, Dong et al. (2018a) proposed an approach that, apart from detecting on-click and placement fraud, also detects if the adware implement procedures for tricking users into unintentionally clicking ads while they are interacting with the UI elements. They present eight rules for identifying fraudulent behavior in adware. These rules were derived from a study of known adware operations as well as from the policies enforced by ad networks. In their proposed approach, snapshots of UI and network traces are recorded while the app is executing. They studied the recorded data to identify violations of these eight rules. Another malicious operation that adware may perform is to redirect users to web sites that contain malware of phishing. Rastogi et al. (2016) studied android apps that lead users the websites that host malicious operations through web links embedded directly in applications or via pages of advertisements that originate from ad networks. They developed an approach that detects these web sites and analyzed their content. Their study resulted in a number of findings that characterize such malicious web sites. For example, they find that a large number of malicious web sites deceive users by claiming to give away free products.

4.3 Methodology

Figure 4.1 shows an overview of our methodology, highlighting the process for extracting information from the dataset to address each of the five research questions. For RQ1, we used the information that already exists in the AndroZoo dataset. We also used the tool Androguard (Desnos 2015) to extract information from the apps' Androidmanifest.xml file. Some of the information extracted includes the app name, name of components and the app's permissions. We also used Androguard to perform a static code analysis to extract the advertisement libraries present in each app. In answering RQ2, we needed to identify the parts of the code that had been manipulated in repackaged apps by comparing the code of repackaged apps with that of the original version. The process needed to accomplish this task is detailed in section 4. After finding the manipulated part of the code in original and repackaged apps, we performed a static analysis using Soot (Bartel et al. 2012) to extract API calls present in the manipulated part of the code. For RQ3 and RQ4, we used the information extracted from Kaggle (Leka 2016) as well as the obfuscation information got through static analysis. We also studied the name of apps extracted from AndroidManifest.xml file of apps. For RQ5, we used the information obtained from the AndroidManifest.xml file of each app.

The dataset used in this study is based on AndroZoo (Li et al. 2017a), one of the largest datasets of Android apps. It was collected from various sources, including the official Google Play app market. AndroZoo currently contains more than 5 million different APKs. Each app is scanned by at least ten different anti-virus products and the results of these scans are reported in the dataset.

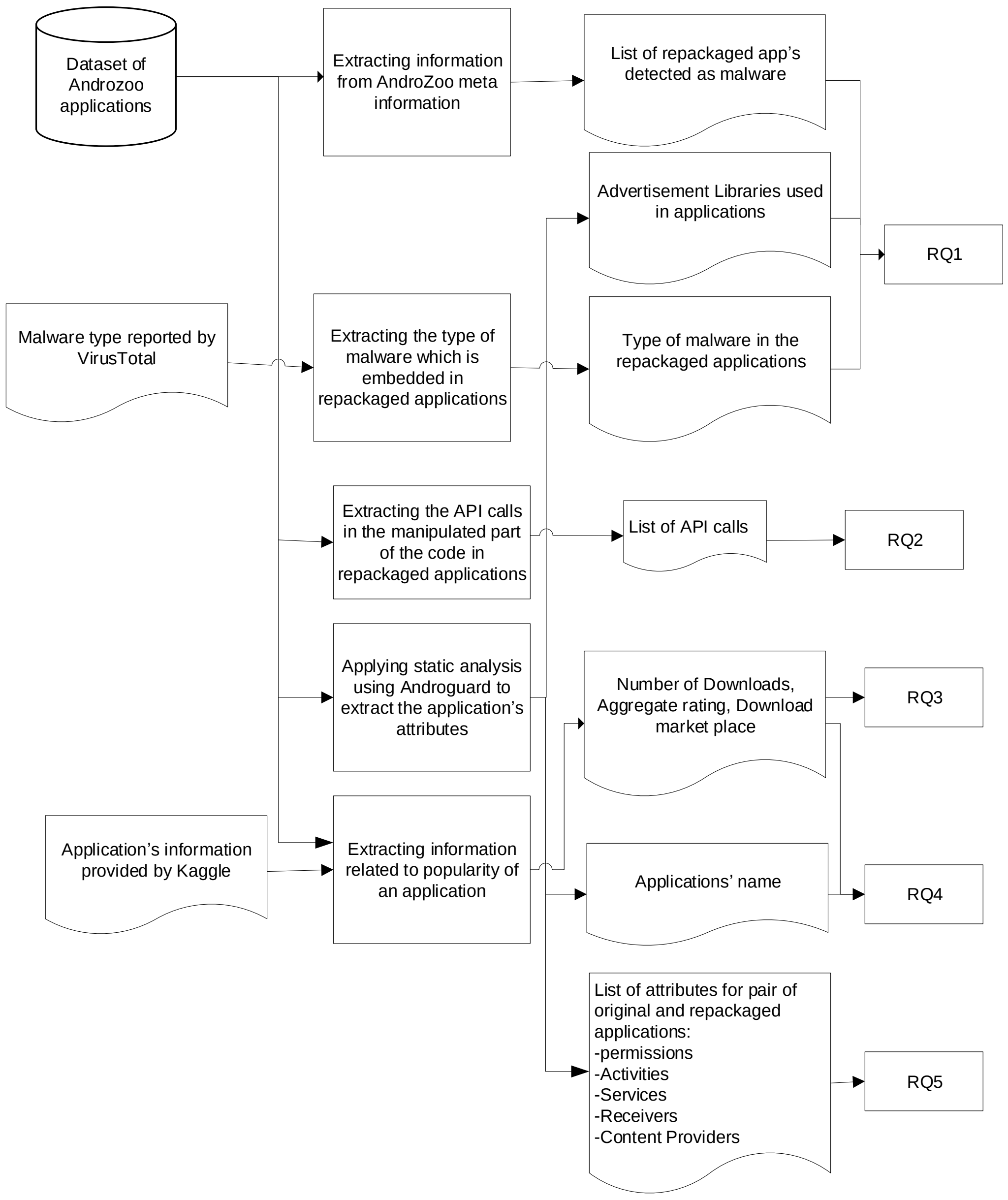

Dataset of Androzoo applications
Extracting information from AndroZoo meta information
List of repackaged app's detected as malware
Advertisement Libraries used in applications
RQ1
Malware type reported by VirusTotal
Extracting the type of malware which is embedded in repackaged applications
Type of malware in the repackaged applications
Extracting the API calls in the manipulated part of the code in repackaged applications
List of API calls
RQ2
Applying static analysis using Androguard to extract the application's attributes
Number of Downloads, Aggregate rating, Download market place
RQ3
Application's information provided by Kaggle
Extracting information related to popularity of an application
Applications' name
RQ4
List of attributes for pair of original and repackaged applications: -permissions -Activities -Services -Receivers -Content Providers
RQ5

Figure 4.1 Overview of our methodology

AndroZoo contains a list of pairs of original and repackaged apps. Each row in the list is described with an SHA (a unique key belonging to each app) of the original app, followed by the SHA of the repackaged app. AndroZoo contains 15,296 repackaged apps of 2,776 original apps, organized in pairs. One original app may have multiple repackaged versions. We used these original-repackaged app pairs as the dataset for our study.

Table 4.1 An example of an app's attributes

Attribute Name	Attribute Value
SHA256, SHA1, MD5	00FAFD8DCDBD59FF035117FF1E19C8329A92AB797E452304FBE82AA9027ACF24,E85B066F301ADD14BED013DB463EE5850316552D, D5AD33B451B84BEF59DA4CB80164544D
APK size	1375601
Market	Slideme
Package Name	kr.mobilesoft.yxplayer
VT Detection	0
VT Scan Date	2013-07-17 20:17:35

An AndroZoo app is described according to the following attributes (see Table 4.1 for an example):

- App identifier: The app is identified using SHA256, SHA1, and MD5 hash values.
- File size: The size of the APK file in bytes.
- Market: The market where the app is published. An app may be published in more than one market.
- Package name: The name of the Android app package, as reported in the manifest file.
- Version code: The app version number, as reported in the manifest file.
- VT Detection: The number of anti-viruses from VirusTotal (2018) that detect malware in the app's code.
- VT Scan date: A timestamp that indicates when an app was scanned by VirusTotal.

In addition to the data provided by AndroZoo, we also used a database of about 300,000 apps, gathered by Leka and made available on Kaggle (Leka 2016). This database contains general information about Android apps including the ones used in this study, gathered from Google Play Store. The information provided includes the APK name, APK file size, price, number of downloads of a given app, and the average rating. We used Kaggle to extract supplementary information about the apps in our dataset when this information was not provided in AndroZoo. The information available in Kaggle includes (see Table 4.2 for example):

- Name: The app's name as reported in the manifest file.
- Number of downloads: Google Play Store provides a range of downloads. In Kaggle's dataset, only the minimum number of downloads is provided.
- Aggregate Rating: Users can rate apps using 1 to 5 stars. The aggregate rating is the average of the ratings assigned by all users.

Table 4.2 An example of an app's attributes in the Kaggle database

Name	Yxplayer
Minimum number of downloads	1,000,000
Aggregate rating	3.335

Finally, we identified the parts of the code that have been manipulated in repackaged apps by comparing the code of each repackaged app with that of its original pair. The details related to the manipulation of the code are explained later in RQ2. After identifying the manipulated part of the code in the original and repackaged apps, we performed a static analysis using Soot (Bartel et al. 2012) to extract API calls from the manipulated part of the code.

4 Empirical Study

In this section, we present, for each research question, the motivation behind the question, our findings and a discussion placing our findings in the broader context of app security.

4.4 RQ1. What is the unfavorable prevailing usage of repackaging?

4.4.1 Motivation

Previous studies (e.g., (Zhou and Jiang 2012b) and (Zhou et al. 2012b)) have identified three main motivations for app repackaging, namely sharing paid apps with no cost, spreading malware, and embedding advertisements. These studies, however, did not provide any statistics that would permit estimating the relative prevalence of each of these motives. Since every app in our dataset is free, we cannot glean any insights related to the prevalence of repackaging for sharing paid apps. We will therefore only focus on malware spreading and advertisement manipulation.

4.4.2 Approach

Each app in AndroZoo is scanned using VirusTotal, a powerful tool that scans apps using more than 30 anti-viruses. The number of anti-viruses that classify an app as malware is included in the AndroZoo dataset. We used this information to determine the number of repackaged apps that introduce malware. To study the presence of advertisements, we compared the advertisement libraries used by the original apps with those present in their repackaged counterparts. We achieved this by applying static analysis over the source code using Androguard. We verified the list of extracted advertisement libraries by cross-referencing them with the ones described by Book et al. (2013). These authors provided a detailed list of advertisement packages commonly used in Android app development. Moreover, we retrieved the name and type of malware in repackaged apps from the work done by Hurier et al. (2017). Their results clarified the percentage of repackaged apps identified as adware.

4.4.3 Findings

F1: 52.22% (7,988 out of 15,296) of repackaged apps are classified as malware by at least two anti-viruses according to VirusTotal reports. All of the apps in our dataset are classified as malware by at least one anti-virus. This is the criterion for inclusion in the dataset used by Li et al. (2017a). Since in some studies (e.g., (Arp et al. 2014) (Canfora et al. 2013)), only the apps detected by at least two anti-virus products are used in their malware sample, we provide data about the distribution of apps detected as malware in our dataset by the number of anti-viruses detecting them in Figure 4.2. As can be seen, 52.22% of repackaged apps are classified as malware by at least two anti-viruses. This suggests that even by the stricter standard used by the papers mentioned above, over 50% of repackaged apps contain malware. Figure 4.3 shows the distribution of repackaged apps, all containing malware, between the years 2010 and 2014. As can be seen in that Figure 4.3, the prevalence of malware in repackaged apps seems to be increasing. While this increase may be partly attributed to the manner in which the data was collected, it does confirm the findings of Zhou and Jiang on this topic. (2012). In 2012, they showed that repackaging is a common vector of malware distribution, based on a dataset of 1,260 apps. Figure 4.3, which shows that the incidence of repackaging in the Androzoo dataset, provides further indication that this trend continued after 2012. Moreover, as shown in Figure 4.4, these repackaged apps are often also published in trusted app stores such as Google Play Store.

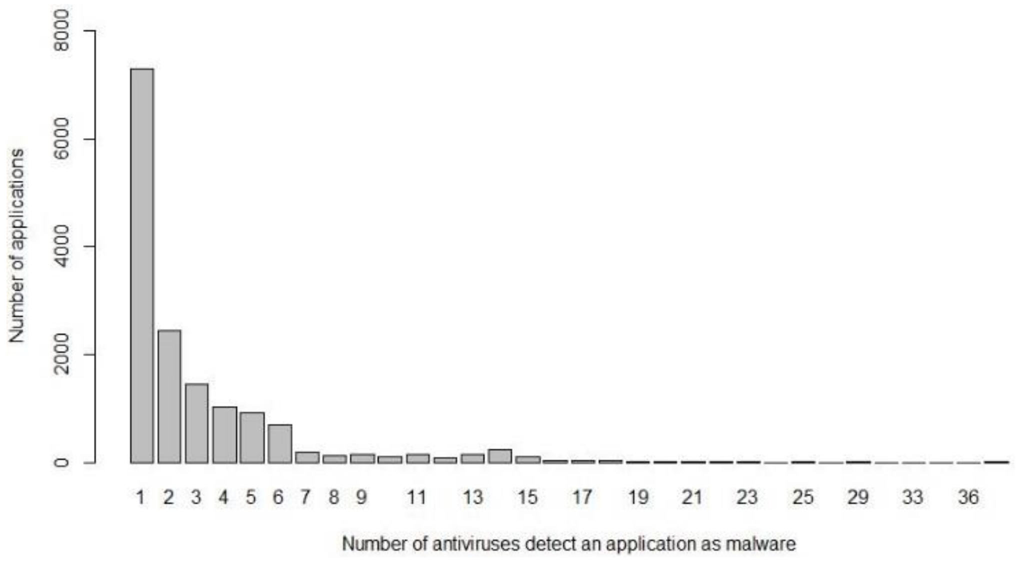


Figure 4.2 Distribution of apps by the number of anti-viruses that identify them as malware

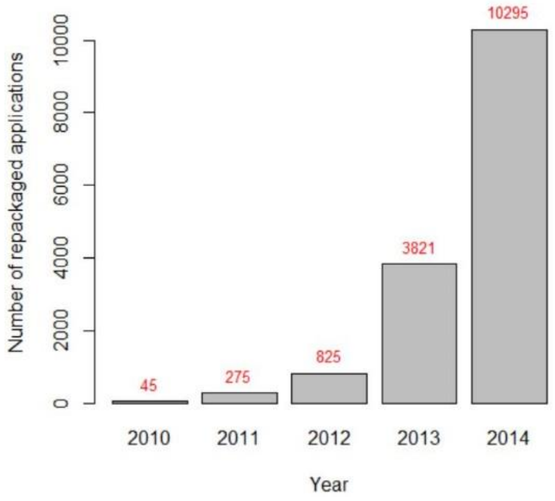


Figure 4.3 Distribution of repackaged app containing malware between the years 2010 and 2014

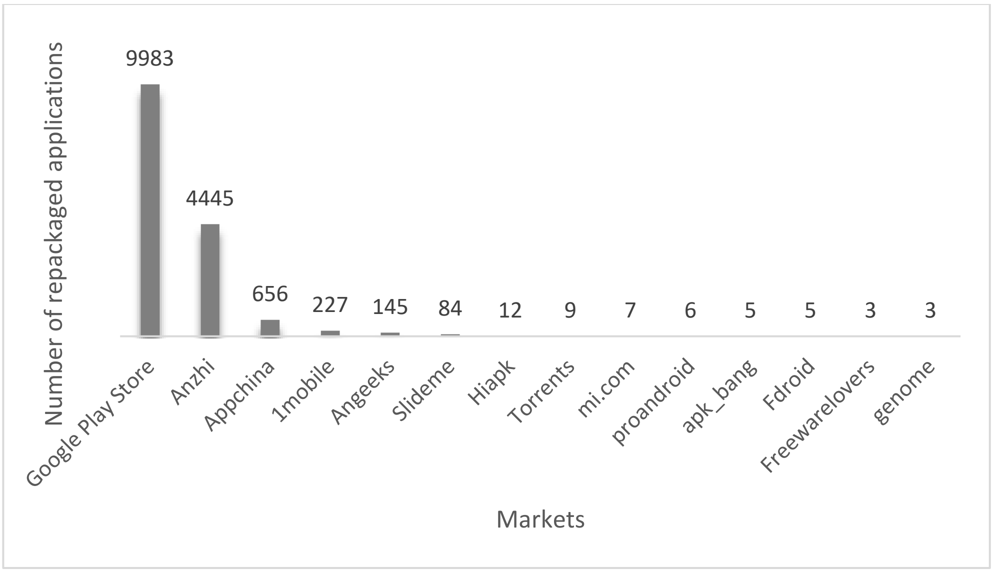


Figure 4.4 Distribution of repackaged apps in a variety of markets

F2: 15.47% (2,367 out of 15,296) of the repackaged apps contain additional advertisement libraries in comparison to their original apps. In 1,968 of these (i.e., 83%), additional advertisement libraries were added to apps that originally had advertisement libraries. We observed that only 399 repackaged apps out of 2,367 (i.e., 17%) had advertisement libraries added while the original versions did not contain any advertisements at all.

In addition, we examined the advertisement packages used in repackaged apps as well as original apps (see Figure 4.5). The data is sorted according to the difference between the frequency of occurrence of each library in repackaged apps and in original apps. We found that com.revmob, com.google.android.gms.ads, and com.mobclix are the advertisement libraries that are used most frequently (in 80% of all cases) in the repackaged apps. Indeed, these specific libraries are also much more likely to be used in repackaged apps than in the original ones.

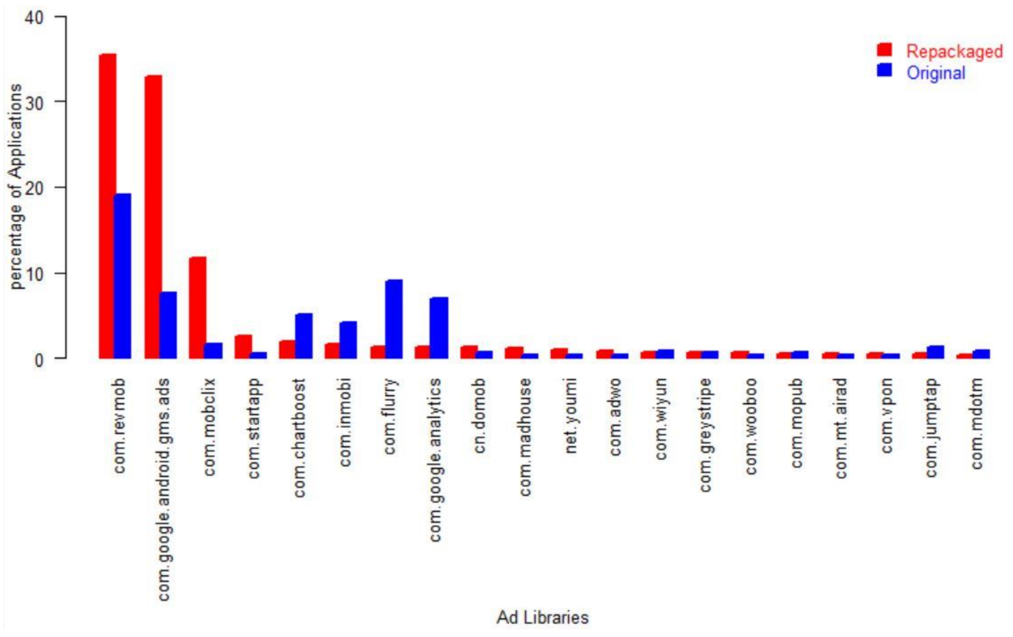


Figure 4.5 Advertisement libraries used in repackaged apps

F3: 77.84% (7,954 out of 10,218) of repackaged apps contain malware of the type "Adware". To identify the type of malware present in repackaged apps, we relied upon a study by Hurier et al. (2017), in which the authors developed an approach to identify the type of malware embedded in repackaged apps. Their approach is based on text mining of VirusTotal reports. However, we were only able to uncover the type of malware for 10,218 of the repackaged apps present in our dataset. We believe that this is because of the assumptions made about the patterns for the malware label which is usually reported in anti-virus's reports. It may happen, for instance, that some of the reports provide a label which does not match with any of the patterns accepted by the tool (Hurier et al. 2017). In some cases, the report may contain the name, but not the type, or vice versa. Here, we considered all the apps for which we were able to obtain both the name and the type of malware it contains.

We found that 7,954 out of 10,218 (77.84%) repackaged apps have malware of type "Adware". We also found that only 1,691 out of 10,218 (16.54%) repackaged apps have malware of type "Trojan". The remaining apps (5.62%) have other types of malware such as backdoors, spywares, worm, etc. Li et al. (2017b) used the most frequently appearing names in the reports of anti-virus engines in order to classify the malware types. They have shown that 888 out of 1,575 (56.38%) apps in their dataset are adware. Our result shows a higher percentage.

4.4.4 Analysis, discussion, and implication

All of the repackaged apps in the AndroZoo dataset are detected as malware by at least one anti-virus and 52.22% are detected as such by at least two anti-viruses (the standard used in multiple studies). It shows the bias in the dataset that focuses on repackaged apps that are detected as malware. Even so, as shown in Figure 4.2 and Figure 4.3, the considerable number of repackaged apps and their increasing trend suggest that repackaging is a common way to distribute malware even in trusted stores such as Google Play Store. F2 and F3 indicate that the repackaged apps are widely used to spread a specific form of malware, namely adware. This finding has implication for malware detection since adware often differs from malware in ways that make detection more difficult, as we will show later in this chapter (Finding F6). As a consequence, we have therefore focused this study on identifying the parts of the code that are manipulated when adware is injected in repackaged apps. A careful study of adware, and of how it differs from other classes of malware, will guide the creation of more effective methods of malware detection.

4.5 RQ2: How is the code of original apps manipulated to embed adware?

4.5.1 Motivation

Findings in RQ1 have shown that a large number of repackaged apps are adware which is distributed in trusted app stores such as Google Play Store. The approaches proposed in the literature focus on detecting malware without differentiating adware from other types of malware. Mariconti et al. (2017) proposed an approach called MaMaDroid to detect malware. Interestingly, amongst the apps not detected as malware by MaMaDroid 45% were adware. Therefore, providing an effective approach for detecting adware is an important challenge. However, in order to effectively detect adware, we first need to study the code of repackaged apps and examine thoroughly how the code in adware components is changed.

The malicious operations in Adware may be limited to showing advertisements in a way that is not concordant with the Advertisement Network policies (e.g., AdMob (2019)) or may include other malicious operations such as reading the device IMEI, the user account list and the device's location and sending this information to a third party. For example, the listing below, in Figure 4.6, shows a segment of code from an adware called "Appenda" (Symantec 2014). This adware contains a service called AppNotify. The service is started as soon as the system is booted. AppNotify pulls ads and show them to the user as notifications. This behavior contravenes to the ad network policies, which states that advertisement can only be shown to users in the apps' user interface, and only while the application is running. Note that out of a concern for clarity and concision, we edited the code by removing some condition checking related to time and network status.

```
public class AppNotify extends Service
{
    public void onStart(Intent intent, int i)
    {
        slinger = new Appenda(getApplicationContext());
        slinger.activateAlarmNotifications();
    }
    private Appenda slinger;
}

public class Appenda //class for pulling and showing advertisements
{
   Appenda(Context c){
    currentServer= (Appenda)com/appenda/Appenda.getClassLoader().
                    loadClass("com.appenda.AppendaServer").newInstance();
    currentServer.setCurrentContext(c);
    if (Condition) //check the time and network availability and update
                   //data frequently
         updateVersion(true);
    SharedPreferences = getCurrentContext().getSharedPreferences(SETTINGS_FILE,
                         0);
    currentServer.setApp_id(getApp_id(sharedpreferences));
    // the data was already saved in a SharedPreference
    currentServer.setPublisher_id(getPublisher_id(sharedpreferences));
    currentServer.setSubid(getSubid(sharedpreferences));
    currentServer.setPublisher_key(getPublisher_key(sharedpreferences));
   }

   public void activateAlarmNotifications()
   {
     while(!isNetworkAvailable() || currentServer == null)
         return;
     currentServer.displayWebAd();//show the advertisement
     updateVersion(true);

   }

   private void updateVersion(boolean flag)
   {
     if(isNetworkAvailable()) {
       SharedPreferences sharedpreferences =
               getCurrentContext().getSharedPreferences(SETTINGS_FILE, 0);
        android.content.SharedPreferences.Editor editor =
               sharedpreferences.edit();
                // read information and save in a sharedPreference
     }
      String phoneNumber= "";
      if(!DEBUG)
           phoneNumber= ((TelephonyManager)getCurrentContext().
                        getSystemService("phone")).getLine1Number();}
      else
        phoneNumber = "8885550001";
        editor.putString("phone_number" , phoneNumber);
        editor.commit();
      }
   }
}
```

Figure 4.6 Snippet code from adware Appenda

4.5.2 Approach

In order to study parts of code that are manipulated when adware is embedded into repackaged apps, we first need to identify and extract such code fragments. To accomplish this task, we relied upon the approach presented in Figure 4.7.

We first grouped the repackaged apps that contain adware based on the name and type if the present, such that two apps containing the same malware type and name (for example all apps with the malware called *Plangton* and whose type is *addisplay)* will be in the same group. In total, there are 106 groups of repackaged apps with different adware name and types. To avoid repetition, we randomly chose only one pair of original and repackaged apps from each group. This is because the changes that occur after a given adware is injected in a repackaged app tend to always the same, for all apps that receive this same adware. Furthermore, sine the number of apps in groups varies greatly, if we considered all apps equally, our results would be skewed towards the changes associated with specific adware, rather than represent a more systematic overview of the types of changes that can occur when adware is added to repackaged apps.

In order to study the manipulated part of the code in repackaged apps that are categorized as adware, we need to extract those parts from the code. To this end, we compared the code of each repackaged app with that of the original app. Figure 4.7 shows the approach we used to achieve this task. First, we used the tool “Dex2jar” to extract the jar file from the dex code of the app. Second, we used a Java decompiler to retrieve the Java source code from the jar file. After obtaining the Java code of every repackaged and original apps, we used piece-wise hashing to generate a digest of files. Piece-wise hashing is a fuzzy hashing method that divides the content into pieces and makes the hash of each piece in order to compute the final hash. Piece-wise hashing provides an almost similar hash if the content of two files have only minor changes, but generates more different hashes if the compared files contain substantial differences (Kornblum 2006). Using this technique, we are able to find pairs of similar files in the code of original and repackaged apps. To hash the Java files of an app, we used Ssdeep (Kornblum 2006), a piece-wise hashing tool developed by Kornblum. Note that it is not feasible to simply rely upon the class file names because, in some cases, the names of the files are altered during the repackaging process, perhaps by using obfuscation tools.

An alternative option to piece-wise hashing is to use code-based similarity techniques. However, it has been shown that code similarity incurs higher time complexity (Huang 2008). To do the comparison in a reasonable time period, piece-wise hashing has been proposed as a data reduction technique that limits the comparison of the content of the entire file to a geenrated hash fingerprint (Kornblum 2006; Li et al. 2015).

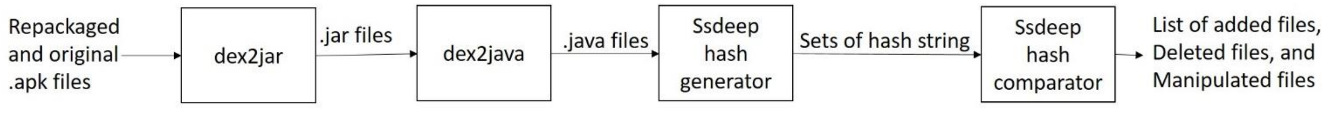


Figure 4.7 Approach for finding the manipulated part of the code in repackaged apps

At this stage, we mapped the apps to a set of hash strings, where each string is the hash of a Java file in the app. We then used Ssdeep to find the files that have been manipulated in the repackaged apps.

Ssdeep computes the match score between two hash strings and returns a number between 0 (no similarity) and 100% (full similarity). We compared each hash string in the set of hash strings of a repackaged app with all other hash strings in the set of hash strings of its original app (the one that is identified as the original app of this repackaged app in the AndroZoo dataset).

More precisely, consider an original app, O, and its repackaged version, R. Let us assume that the number of files in the repackaged app is M and the number of files in the original app is N. Using Ssdeep, we compute the match score between each two files, f_i and b_j of R and O, respectively with i:1..M and j:1..N. The objective is to identify:

- Manipulated Files: The files in the repackaged app, R, which are manipulations of files in the original app, O.
- Added Files: The files in the repackaged app, R, that do not exist in the original app, O.
- Deleted Files: The files in the original app that were removed from the repackaged app.

To achieve this, we use two thresholds t1 and t2 (t1 < t2):

- If a hash string of a file in the repackaged app matches a file in the original app with a score of t2 or higher then we consider the two files as identical in both R and O. The underlying classes require no further study and we remove them from both sets.
- If a hash string of a file in the repackaged app matches a file in the original app with a score between t1 and t2 then we conclude that a file has been manipulated. We will examine such files later on.
- If a hash string of a file in the repackaged app does not match any hash string of any file in the original app with a score above t1, then the mapped repackaged app file is considered as a file added in the repackaged app.
- If a hash string of a file in the original app does not match any hash string of any file in the repackaged app with a score above t1, then the mapped original app file is considered a file deleted after repackaging.

We determined t1 and t2 through experimentation. We varied t1 from 10% to 100% with a step of 5% and found that when setting t1 to 70%, we obtain at most one file of the repackaged app that is a manipulation of a file of its corresponding original app. This threshold was also used by Zhou et al. (2012a) when detecting similar apps for identifying repackaged apps in third-party marketplaces. As for t2, we opted for 98% similarity because we found that it is the highest value to correctly match classes of the app files while being tolerant to very minimal alterations such as changing the name of the class. A lesser threshold could be used, in which case, we may end up excluding files in the repackaged apps that are manipulations of files in the benign apps. To reduce this risk we decided to keep 98%.

After obtaining a list of files, which have been manipulated, deleted, or added, we performed a static analysis using Soot (Bartel et al. 2012) to extract API calls in the manipulated part of the code. Soot allows us to avoid the drawbacks of using tools for decompiling a dex code to its Java code by providing static analysis over the dex code directly. In particular, Soot can generate a listing of all API calls present throughout an app's code. Using this functionality, we listed API calls present in each class in pairs of original and repackaged apps and compared them. For the manipulated files, after extracting the API calls, we used the module "sets" in Python to extract the different API calls in pairs of original and repackaged manipulated files. In the manipulated files, the subset of API calls that exists in the original app but not in the repackaged ones corresponds to deleted API calls. Conversely, a subset of API calls that exist in the repackaged app but not in its original app pair corresponds to added API calls. Note that using this method for identifying differences understates the changes that occur in the repackaging process since some manipulated API calls will be present in both the original and repackaged apps, but with different input parameters or calling context.

The main benefit of the approach we propose is that hashing can be performed over the entire content of the files instead of only on their names, or on the names of the methods and classes. Therefore, even if the name of files and contained classes is changed (a common obfuscation technique), our approach can still determine that the different files are related based on the piece-wise hashing of their contents.

Note that in the presence of obfuscation, an API may appear to have been simultaneously removed from the original app and added to the repackaged app, when in fact the code is unchanged, only obfuscated. This fact does introduce a small possibility of error. However, in our dataset, this scenario is actually rather uncommon. In fact, less than 4% of the repackaged apps (see Table 4) exhibit obfuscation of a type that would introduce this error, such as name changing.

4.5.3 Findings

F4: Some permissions are more frequently requested in adware. Figure 4.8 shows the top 20 most frequently added permissions in adware samples in comparison to the rest of malware. To improve readability, we have removed the first two parts of the permission's name which usually are "android.permission" or "com.android". In total, there are 260 out of 7,954 adware and 95 out of 1,691 samples with other types of malware (i.e., not adware) in our dataset, which contain a set of permissions added in the repackaged version in comparison to its original pair. The frequently added permission "android.permissions.GET_TASK" is used to obtain information about the processes that are executed by apps. It was deprecated in API level 21 due to security violations it may cause. Permission "com.android.vending.CHECK_LICENCE" is frequently used in adware, but no so much in Trojans. Applications need this permission to use a Google service that verifies the license of applications. It provides an infrastructure to allows apps to connect to a distant host, and its presence suggests that the developers of repackaged apps might wish to connect to the devices running their apps for some reason. More research is needed to elucidate the motivation of repackaged app developers in activating this functionality. Permission "com.android.vending.BILLING" is used for purchasing in-app products such as additional game levels, media files, or online magazine services. It has been used in 17 out of 260 adware samples. Obviously, if an app has this permission, it can also use it to perform payments not authorized by the user. Some studies (Mulliner et al. 2014, Raynoud et al. 2012 and Dong et al. 2018a) focus on the threat related to detouring in-app purchases in order to obtain free services without paying. To the best of our knowledge, there is no study showing that in-app purchase is exploited by adversaries to get revenue. More studies are needed to clarify how this permission is exploited and what are the possible protections against this threat. Another permission commonly used in adware is "android.permission.VIBRATE". It allows access to the vibration setting and can be useful to get users' attention, but can be annoying to some users. The other permissions frequently added in repackaging serve to access device resources, such as Internet connection or logs.

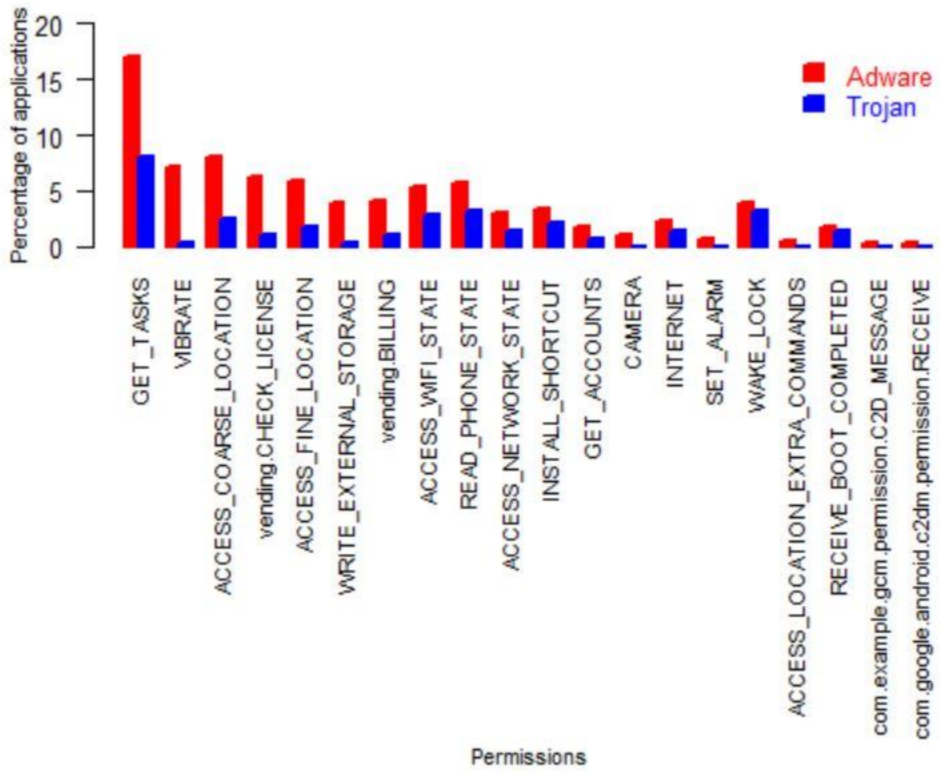


Figure 4.8 The frequency of the top 20 permissions most frequently added to adware as opposed to trojan

F5: In 27% of repackaged apps containing adware with no call to APIs that require permission to execute. We studied the API calls extracted from files added in repackaged apps as well as the APIs present in the manipulated files from repackaged apps but not its original pair. Among the extracted APIs, we identified the APIs that require permissions to execute by relying upon a study by Wain et al. (2012). From 106 adware samples, each selected randomly from each group of repackaged apps with different adware name and type, 26 (24.52%) did not require permissions to run. This suggests that these adware samples do not perform malicious operations such as leaking confidential data or executing commands in devices. We studied the Java code of these samples. We found that reflection calls are present in 5 out of 26 samples in original and repackaged version, but with different parameter strings. Those samples also contain name changing obfuscation. For the rest of the samples, we found that the files changed in the repackaged apps are in fact the obfuscated version of files in the original version. The obfuscation only name changing and changing the name of some of the variables. It seems that the developers of the repackaged app obtain revenue by publishing the app under his own name. There remains the question determining how VirusTotal identified the repackaged app, but not the original app, as adware. While it was not the case in our dataset, there may be cases in which the permissions that are requested at run time and where malicious code is added dynamically at run time. Therefore, using static analysis is not sufficient to detect these kinds of adware.

F6: Detection of adware based on studying only API calls may not be effective. In Figure 4.9 and Figure 4.10, we used the package of API calls instead of the API calls themselves to present a summary of manipulated API calls that require permissions in order to execute. As shown in Figure 4.9 and Figure 4.10, the most frequently added API calls and most frequently deleted API calls have considerable overlap. In each pair of original and repackaged apps, we extracted the APIs that have been added to and deleted from the original app. We found that that most of the API calls exhibit a similar distribution in original apps and the repackaged adware samples.

Previous research used API calls for malware detection (Alazab et al. 2010; Islam and Altas 2012; Wu et al. 2012; Chen et al. 2015a; Zhou et al. 2012a; Enck et al. 2014; Grace et al 2012; Mariconti et al. 2017; Aafer et al. 2013; Yang et al. 2014). In these papers, the effectiveness of the detection is evaluated using apps randomly selected from app stores. We should, however, raise two important points with respect to these investigations. First, none of these methods is based exclusively on the frequency of API calls. For example, Aafer et al. (2013) used other features including parameters passed to API calls, or Mariconti et al. (2017) used a call graph of API calls. Furthermore, it is still unclear how well they will work when the testing set contains the original version of repackaged apps. Indeed, as shown in Figure 4.9 and Figure 4.10, highly frequent API calls in added files in repackaged apps are similar to those in deleted files in original apps. Therefore, the accuracy of any proposed malware detection based on API calls must be tested with dataset that contains matching pairs of original as well as repackaged apps.

Finally, note that the literature does not differentiate between adware with other types of malware; our finding F3 shows that adware samples form a large portion of malware (77.84%). Therefore, the similarity of API calls in added and deleted files described above may occur in large portion of malware of type adware.

4.5.4 Analysis, discussion and implication

Findings F4, F5 and F6 further suggest some additional assertions about adware. First, permissions relating to in-app purchases suggest that adware may generate revenue for their creators through the use in-app purchases, especially since every repackaged app in our dataset is free. Second, since API calls deleted from original apps and API calls added in repackaged apps exhibit a very similar distribution, detection of adware based only on the distribution of API calls seems impractical. Third, as shown in Figure 4.10, some API calls, present in the files that are deleted from the original apps, are related to the connection to a remote server. This confirms the expectation that adware developers need to manipulate those API calls or delete them. However, we leave a complete study of the parameter of API calls for future work. It is very promising to understand to what extend the value of parameters in API calls are changed during repackaging. Fourth, the deleted API calls include API calls that establish a connection to a remote server and in doing so might reveal the origin of the app. In this regard, any kind of code obfuscation that hinders code comprehension can be useful in preventing repackaging. Since we observed in F5 that a considerable proportion of adware does not require permissions to perform their operation, it is not practical to rely upon permissions for adware detection. Finally, since the similarity of API calls in manipulated part of the code in adware samples and their original version is very high, we also suggest that the developers of adware detection methods test their approaches' accuracy over a testing dataset that contains both original and repackaged version of the malware. However, an effective detection could be performed by recognizing these apps as repackaged clones of other available apps. In RQ5, we propose an app classification scheme that allows the detection of clone apps to be performed in tractable time.

4.6 RQ3. Which types of apps have been exploited for repackaging?

4.6.1 Motivation

Previous literature identified the popularity of an app as the main criterion for an app being chosen for repackaging and for spreading malware (Li et al 2017b). Hence, we examined the popularity of the apps exploited for repackaging in our dataset.

Apart from the popularity of apps, we also sought to shed light on code comprehension, as it applies to embedding malware into apps. Obfuscation tools, such as Dexguard[2], can make it harder to understand the code, for example by changing the name of methods and variables and by using reflection. There is also the question of whether one needs to understand the code to manipulate it. We studied the use obfuscation in the apps of our dataset to determine if its usage negatively correlated with repackaging.

[2] http://www.guardsquare.com/en/dexguard

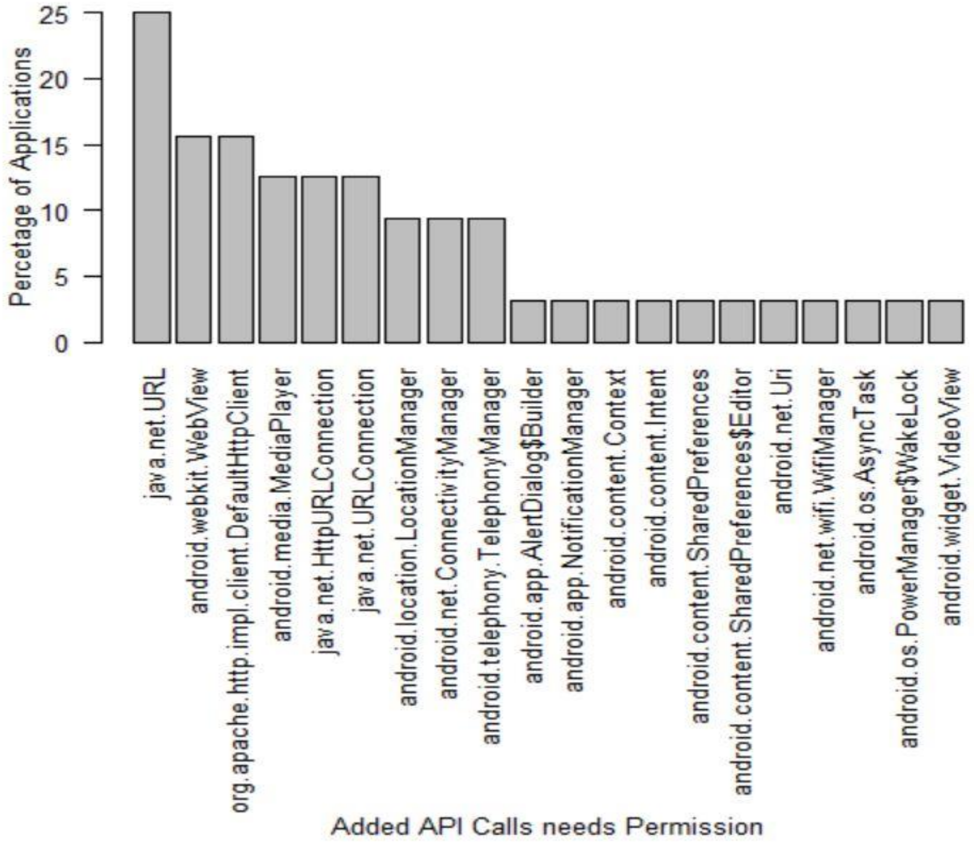


Figure 4.9 Top 20 frequently APIs requiring permissions to execute, added in repackaged adware samples in comparison to their original apps

4.6.2 Approach

We defined three metrics by which to measure the popularity of an app: the star rating of the app, its number of downloads, and the market where the app is located.

In the Google Play Store, users can rate apps by assigning them from 1 to 5 stars. The average number of stars is called the aggregate rating and it is recorded in the app's page in the Google Play Store. We used the aggregate rating of each app as our first metric of popularity. We were unable to extract this information directly from the Google Play Store because many apps in our dataset are no longer available. However, as mentioned in Section 3.3, this information was available in the Kaggle dataset (Leka 2016) gathered in May and June 2014, but, unfortunately, for only 686 original apps out of the 2,776 original apps in our dataset.

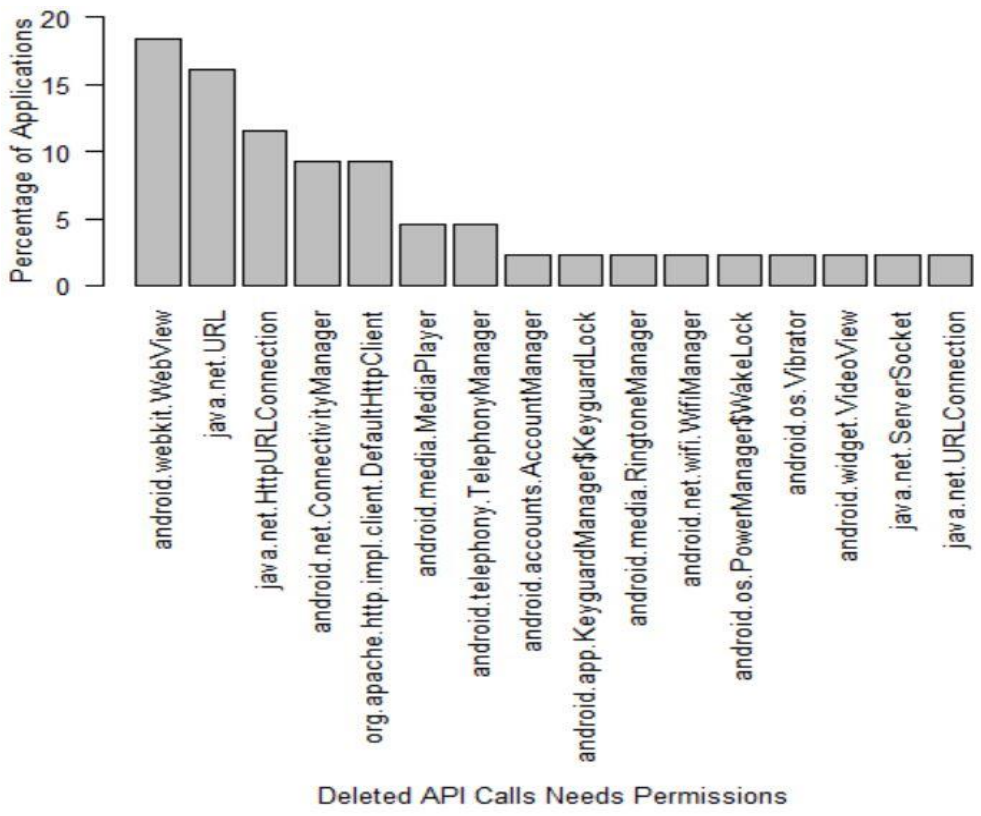


Figure 4.10 The top 20 frequently APIs requiring permissions to execute that are deleted in repackaged adware samples in comparison to their original apps

We used the number of downloads as a second metric for measuring the popularity of apps. The third criterion is the market where the original apps are located. This information is provided by AndroZoo for all 2,776 original apps. Note that an app may be published in more than one market. We considered every market in which an app is published in our statistics. For example, if an app is available in the Google Play Store as well as in Anzhi, we count it once in the Google Play Store and once in Anzhi.

Finally, we studied whether obfuscation (name changing, reflection, and dynamic loading) is present in the apps. The most popular obfuscation tools rename program methods using a simple alphabetic change. Methods are first renamed with a single letter such as *a*, *b*, *c*,…. When the alphabet is exhausted, the algorithm proceeds with two letter names such as *aa*, *ab*, ... and so on. We used Soot to perform static analysis over the dexcode of apps to find evidence of obfuscation, such as names in the above-described format. The presence of methods using the class “java.lang.reflect” and “dalvik.system”, which indicate the presence of reflection and dynamic loading in the app code, respectively, are taken as evidence of obfuscation. The later assumption is justified by the fact that according to the Android developer documentation (2018), every class of the dalvik.system implements a classloader or some classloader functionality. Several other papers identify dynamic loading as the purpose of the dalvik.system class (see for example (Maly and Kriz 2015), (Zhao and Qian 2018), (Poeplau et al. 2014), and (Zhou et al. 2012b)).

We compared the results for the original apps with the information obtained for all other free apps available in Kaggle dataset. Kaggle provides information including app's name, publication date, file size, star rating, number of downloads, package name and price. There are 245427 free apps (with price 0.0). We examined free apps exclusively because all the repackaged and original apps in our dataset (AndroZoo dataset) are also free.

4.6.3 Findings

F7: Apps with higher star ratings are not more likely to be repackaged. Figure 4.11 presents the boxplot of the aggregate star-rating for original apps that have been repackaged compared with that of the free apps in Kaggle dataset. To mitigate the threat of time inconsistency, we grouped the apps by year of release date, since apps that have been in a store longer time are likely to have received a longer number of user reviews. This comparison reveals that median aggregate star-rating for both is almost identical. However, the aggregate star-rating of original apps varies in a much narrower range than that of all apps. However, the Wilcoxon-Mann-Whitney test only shows a significant difference for the aggregate rating between original apps and all free apps (with p-value=0.03054) in 2012. It does not show a significant difference for apps in 2013 and 2014 because the p-value is above 0.05. These results do not fully support previous research that indicated that popular apps are more likely to be repackaged (Li et al 2017b) based on the metric aggregate star-rating.

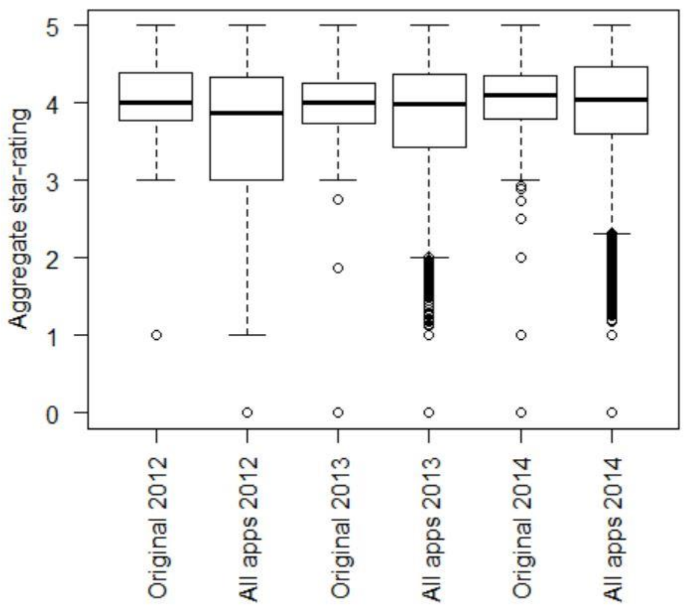


Figure 4.11 Boxplot showing the aggregate rating of the original apps vs all the apps

F8: Apps with a high number of downloads are more likely to be repackaged. Figure 4.12 shows the distribution of original apps versus all free apps according to their number of downloads. To mitigate the threat of time inconsistency, the apps are grouped according to their release date. Figure 4.12 shows that the number of downloads for original apps is skewed to the right, where the number of downloads is increased. But, for all apps, the number of downloads is skewed to the left (where the number of downloads is lower). Moreover, using Wilcoxon-Mann-Whitney test over the set of number of downloads for original apps and number of downloads for all free apps in the same year shows that these two sets are significantly different with the p-value<0.1e-6. These results were expected, and confirm that malware developers target popular apps in order to spread malware.

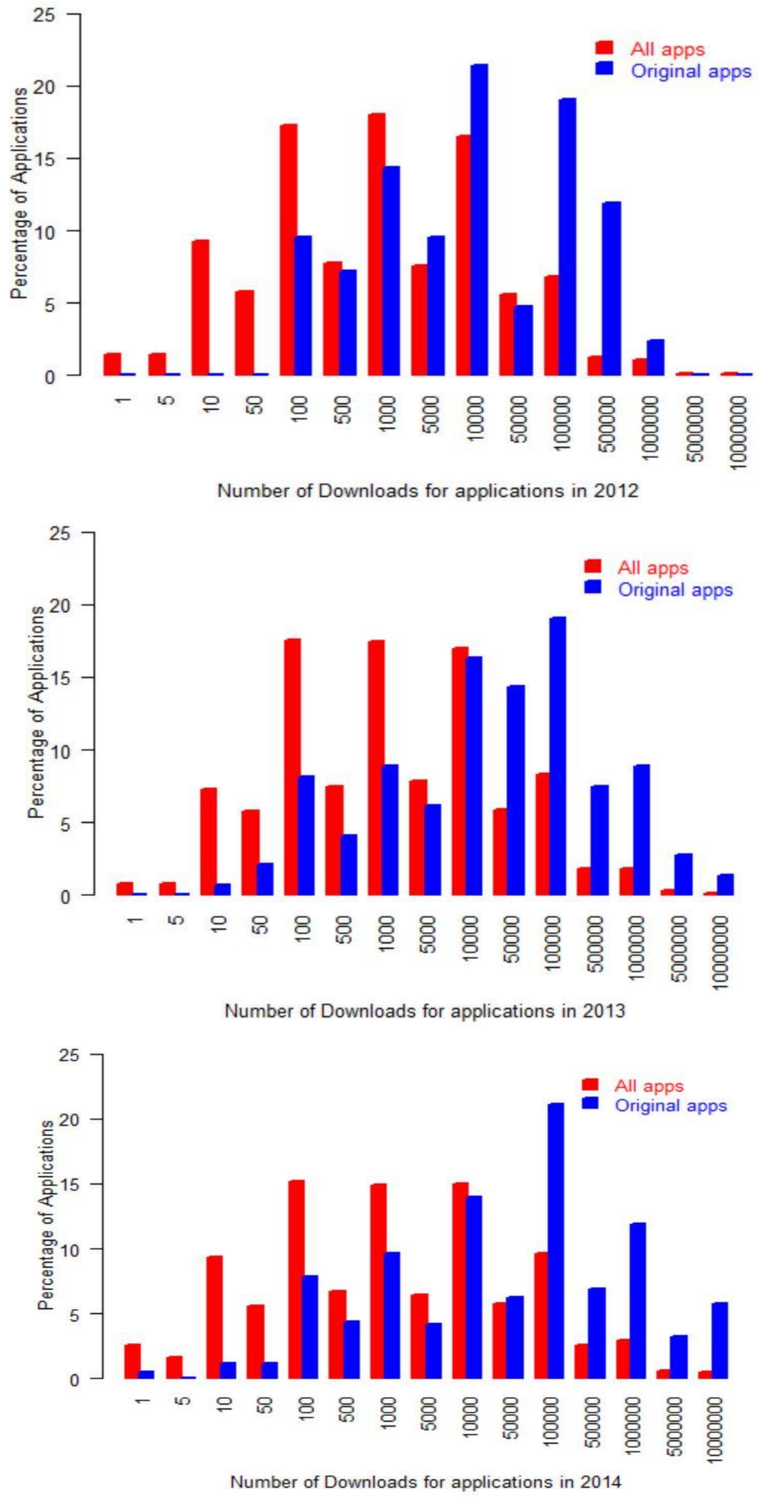


Figure 4.12 Frequency of the number of downloads of apps in a variety of ranges

F9: Apps obfuscated with name changes are less likely to be repackaged. Table 4.4 shows the percentage of original and repackaged app pairs subject to various obfuscation techniques including name changes, reflection, and dynamic loading. The results of reflection and dynamic loading do not seem to yield meaningful information. However, the results related to name-changing are striking. Indeed, 95.94% of app pairs exhibit no name changes between the original code and that of the repackaged app. Moreover, in an additional 0.16% of pairs, the original apps do not exhibit obfuscation though name changing while the repackaged version does. In total, 96.1% of original apps in our dataset are not obfuscated though method name changes. Dong et al. (2018b) studied the obfuscation techniques used in Android apps. We summarized their results in Table 4.3. It shows that 43% of apps in Google Play and 73% of apps in third-party contain obfuscation of type name changing. Therefore, 57% and 27% of apps in Google Play and third-party apps, respectively, are not obfuscated. By using the Chi-square test, the ratio of original apps that are not obfuscated with type name changing is significantly different from the ratio of apps in Google Play with p-value = 2.839e-15. It is also significantly different from third-party apps with p-value=1.268887e-54. This result suggests that the designers of repackaged apps deliberately target apps whose code has not been obfuscated and that such apps are much more likely to be repackaged.

Table 4.3 Obfuscation techniques (Dong et al. 2018b)

App Store Name	Obfuscation	
App Store Name	Name changing	Reflection
Google Play apps	43.0%	48.3%
Third party apps	73.0%	49.7%

F10: Dynamic loading is added during repackaging up to 1/6 (16.46%) of the time. We studied two other types of obfuscation, namely reflection and dynamic loading, and summarized our results in Table 4.4. The first two rows show that the existence of each type of obfuscation in the original and repackaged app. It shows that 16.46% of repackaged apps exhibit dynamic loading while the original app does not (last row). This suggests that the malicious operations may not statically present in the code of the app and are instead loaded later at runtime. This result confirms the importance of dynamic analysis to ensure the security of mobile devices.

Table 4.4 Percentage of apps pairs that exhibit or do not exhibit each variety of obfuscation techniques

	Obfuscation type		
	Name changings	Reflection (%)	Dynamic Loading (%)
Percentage of pairs for which obfuscation is present both in the original and in the repackaged version	3.55	99.09	52.65
Percentage of pairs for which obfuscation is present neither in the original nor in the repackaged version	95.94	0.60	30.56
Percentage of pairs for which the original app exhibits obfuscation but the repackaged version does not	0.34	0.04	0.33
Percentage of pairs for which the original app does not exhibit obfuscation, but the repackaged app does	0.16	0.27	16.46

4.6.4 Analysis, discussion and implication

We studied the popularity of apps based on two metrics including aggregate star-rating and number of downloads. Based on the star-rating presented in F7, we cannot completely support the idea that popular apps are selected for repackaging. But, according to finding F8, the number of downloads in original apps which are selected for repackaging exhibits a statistically significant difference with that for all free apps. We conducted a correlation test (Pearson and Spearman) regarding these two metrics but found a very low correlation between them. This shows that having a high star rating is not necessarily accompanied with having a high download rate. Moreover, among the repackaged apps there are still a considerable number of apps that do not have a high star rating or a high number of downloads. These results have a clear implication for the effort to detect repackaging: such efforts simply cannot be focused on highly popular apps. Furthermore, there is also the question of which available popular apps are more likely to be selected for repackaging. Result F9 show that apps that do not exhibit name changes obfuscation are most likely to be repackaged. Therefore, while the popularity of an app tempts attackers to repackage it, they still prefer apps whose code is not obfuscated. This result reconfirms the importance of obfuscation and of research in improving obfuscation techniques as it relates to protecting the revenue of apps. We believe that using the alphabet for name changing can still reveal information about the target program since there is a defined ordering in which the classes are processed.

F10 presents a supplementary result, not related to the above research question, which confirms that static analysis alone is not enough to detect the presence of malware in repackaged apps, since the malicious elements may be loaded at runtime.

4.7 RQ4: Why users download the repackaged apps when the original version is freely available?

4.7.1 Motivation

To propose approaches to defeat and prevent repackaging, it is useful to know the reasons why users download the fake version of an app even though the original version is available for free. Clearly, users do not know that they are downloading a fake version, especially since they are using trusted stores like Google Play Store.

4.7.2 Approach

In understanding user behavior for deciding to download an app, we must consider a variety of metrics. A complete review of all factors affecting user behavior is beyond the scope of this thesis, which focuses on a statistical analysis of the apps' manipulations. Nonetheless, there are still some findings that can help us to partially answer this research question.

4.7.3 Findings

F11: Repackaged apps originate from a variety of markets including Google Play Store. We found that most of the original apps that have been repackaged were originally published in the Google Play Store. Figure 4.13 shows that 58% of apps were published on this platform. Anzhi, 1mobile and Appchina are the other favorite markets. While the Google Play Store is a popular store for Android apps, repackaging exploiters seem to use it as the main place for browsing and finding target apps to spread malware.

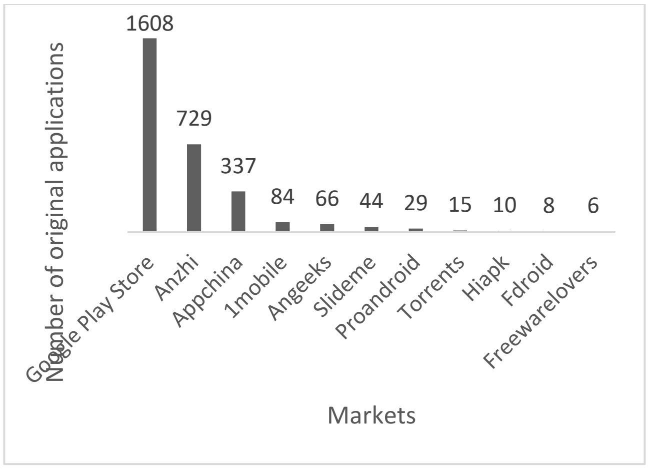


Figure 4.13 Number of original apps according to the market where they are published

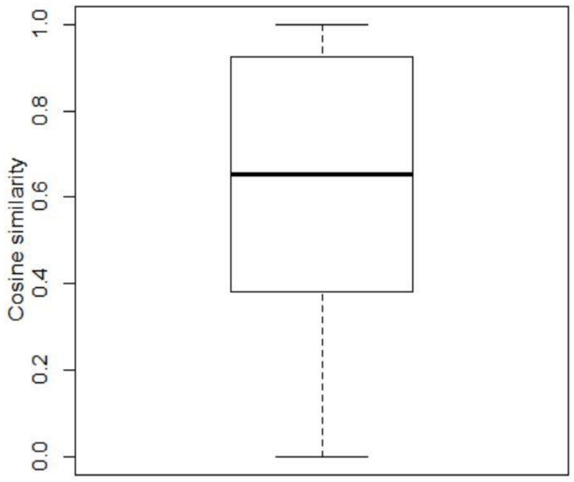


Figure 4.14 Boxplot showing the similarity score between the name of the original app and the name of its repackaged version

F12: A considerable number of repackaged apps are published again in the same store as the original ones. We also find that 13,881 out of 15,296 (90.75%) of repackaged apps were republished again in the same store, and that the number published in other stores is 1709. Note that 294 of these were published in both the same and other stores. Therefore, those apps are repeatedly counted in the number of repackaged apps stored in the same store and the number of apps published in other stores.

F13: Repackaged apps' names are usually very similar to the name of the original app. We extracted the app's name by using Androguard. Figure 4.14 shows a boxplot that represents the cosine similarity measure (Singhal 2001) between the name of repackaged apps and the name of their original app. The figure shows that more than 50% of the original and repackaged pairs have a name similarity greater than 65.30%, which suggests that developers of repackaged apps deliberately maintain similar names to the original apps, perhaps to dupe users. For example, an app whose name was “Duck Shooter” in the original version was repackaged and renamed “Shoot My Duck”. The cosine similarity between these names is 84.27%.

There remains a considerable number of app pairs with different names. In fact, 35.40% of the app pairs exhibit a name similarity of less than 0.5 based on the cosine distance metric. We used Google language detection library in Python[3] to detect the languages in which the name of apps is written and then compared the languages for pairs of repackaged and original apps. In total, there were 796 original apps and 6733 repackaged apps that were in languages other than English. There are 1,164 out of 15,296 pairs (7.60%) where the language of the app’s name in the original and repackaged version is different. This result suggests that the repackaged app’s language may be another motivation for users to download the repackaged version rather than the original. More research is needed in order to clarify the user’s motivation in downloading apps in languages other than English. We could examine, for example, whether the layout of the app catalog or user reviews favor these alternate language apps over their English counterparts.

F14: Repackaged apps in the Google Play Store have a high star rating and a high number of downloads. In the Kaggle dataset, we found the data for 2035 repackaged apps that exclusively come from the Google Play Store. Figure 4.15 and Figure 4.16, respectively, present the aggregate star rating and number of downloads for those apps. Figure 4.15 depicts that 75% of repackaged apps have an aggregate rating greater than 3.66. In addition, 50% have a rating greater than 3.96, and 25% have a rating greater than 4.31. Moreover, using the Wilcoxon-Mann-Whitney test does not support the hypothesis of a statistically significant difference between the star ratings of original apps and that of repackaged apps. Furthermore, as can be seen in Figure 4.16, 58% of apps have more than 1000 downloads. These are considerably high download number and high star rating for repackaged apps and suggests that repackaged apps are popular and as a consequence, that they are successful in spreading malware.

[3] https://github.com/Mimino666/langdetect

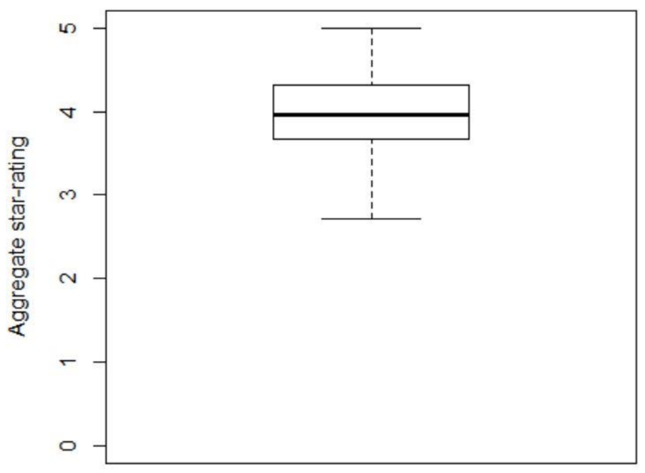


Figure 4.15 Boxplot showing the star rating for the repackaged apps

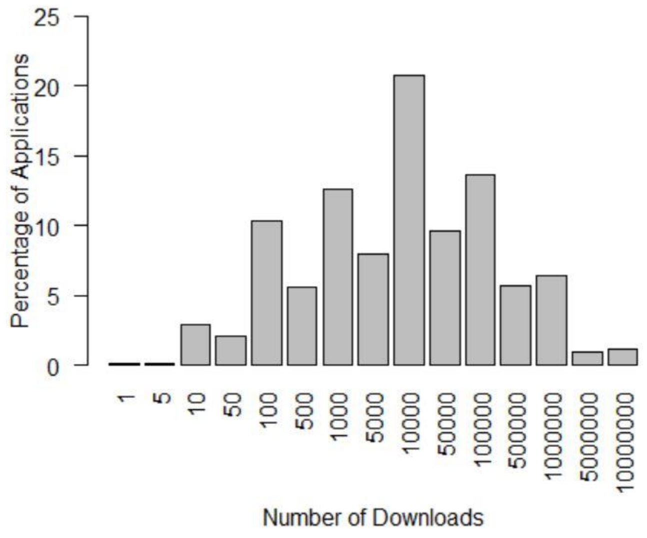


Figure 4.16 Distribution of repackaged apps based on the number of downloads

4.7.4 Analysis, discussion, and implication

Result F11 shows that most repackaged apps were originally downloaded from Google Play Store. This suggests that the repackaging exploiters use the most popular store to obtain a list of popular apps and thus exploit the users' tastes. The interesting point here is that based on F12, the repackaging exploiters target the users of the same store in distributing the repackaged versions. Moreover, result F14 indicates that they are often successful in deceiving users, getting them to download their repackaged apps. On the other hand, according to result F13, there is high similarity between the app name in original and repackaged versions. Similar names increase the chance for the repackaged app to be listed by search engines when a user is looking for the original app. Moreover, the date of the publication of the repackaged version is, obviously, later than its original pair in the same store. These statistics suggest that users may be fooled, assuming that the repackaged app is the latest version or new release of the original app.

For the repackaged apps published in other stores, other than the original version's store, more study is needed before conclusions could be reached. For example, it would be interesting to study the language of the layouts in the app or the language in which reviews are written.

4.8 RQ5. How are an app's attributes modified in the repackaged version?

4.8.1 Motivation

Knowing about the changes in the attributes of an app after repackaging can guide researchers in proposing approaches for detecting repackaging. While investigating this research question, we focus on the following attributes: the APK name, the size of the APK file, the app components, and the list of permissions.

4.8.2 Approach

We extracted an app's attributes from the manifest.xml file of each app and compared the attributes of the original apps to those of the repackaged versions. We compared the number of components in the original and repackaged apps. We also compared the name of each component, using the cosine similarity distance (Singhal 2001) to measure the extent by which the two names are deemed similar. We proceeded in like manner for permissions, i.e., we compared the number of permissions as well as the name of permissions.

We used Androguard (Desnos 2015) to extract the following attributes from each app: app size, app name, the number of components and their names, and the apps' permissions. This data is extracted from the Androidmanifest.xml file that accompanies each Android app and contains metadata about the app. It notably includes the app components' names and permissions. Table 4.5 shows an example of the attributes of an app that are extracted by Androguard from the Androidmanifest.xml file.

Table 4.5 An example of the components of an app

Attribute Name	Attribute Value
Activities	com.ansca.corona.CoronaActivity|com.ansca.corona.CameraActivity,com.ansca.corona.VideoActivity|com.openfeint.internal.ui.IntroFlow,com.openfeint.api.ui.Dashboard,com.openfeint.internal.ui.Settings|com.openfeint.internal.ui.NativeBrowser|com.adknowledge.superrewards.ui.activities.SRPaymentMethodsActivity,com.adknowledge.superrewards.ui.activities.SRDirectPaymentActivity,com.adknowledge.superrewards.ui.activities.SROfferPaymentActivity,com.adknowledge.superrewards.ui.activities.SRWebViewActivity,com.zong.android.engine.web.ZongWebView
Services	com.zong.android.engine.process.ZongServiceProcess
Receivers	com.ansca.corona.purchasing.GoogleStoreBroadcastReceiver
Content Providers	com.ansca.corona.FileContentProvider
Permissions	android.permission.INTERNET|android.permission.READ_PHONE_STATE|android.permission.ACCESS_NETWORK_STATE

4.8.3 Findings

F15: The size of the APK file may or may not increase after repackaging. Figure 4.17 shows a plot of the size of the original app' files in comparison with the file size of the repackaged version. The figure shows that the size may or may not increase after repackaging, suggesting that the APK file size criterion alone is not an indicator of repackageability. Note that a strong similarity is obtained when the (x, y) points of the graph are positioned on the blue line.

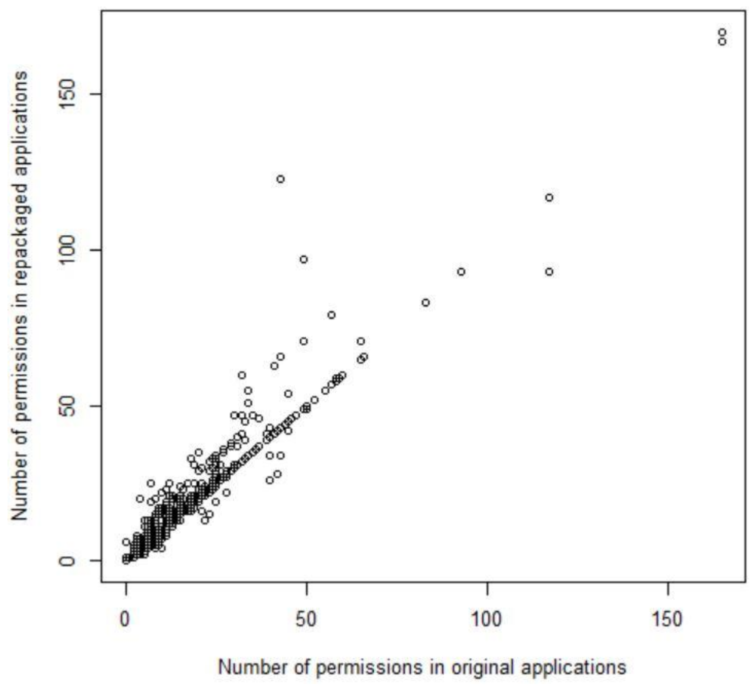


Figure 4.17 Comparison of the APK file size of the original apps with the file size of the repackaged versions

F16: Repackaged apps have a similar number of components in comparison to their original pair. Figure 4.18 shows the number of app components in the original apps compared with that in the repackaged apps. We extracted the number of components from the apps' Androidmanifest.xml file. Note that app developers should write the names of activities, services, and receivers in Androidmanifest.xml file. However, content providers can be created at runtime. We found that the number of activities, services, receivers, and content providers did not change much in the repackaged apps. The number of activities is the same in 14,366 out of 15,296 app pairs (93.92%). The number of services is the same in 15,016 out of 15,296 app pairs (98.17%). The number of receivers is also the same in 15,043 out of 15,296 pairs (98.35%). This also applies to the content providers where the number of content providers is the same in 15,241 out of 15,296 app pairs (99.64%). These results demonstrate that the number of components of an app is not an indicator of repackageability. This information can however be useful in finding similar apps.

F17: Components' name does not change in most of the repackaged apps in comparison to their original pair. We also examined the changes in component names. We found that among 14,366 original-repackaged app pairs with the same number of activities, 11,527 of these (80%) have identical names in the original and repackaged versions.

There are 7,466 original and repackaged app pairs that have at least one service, and that have the exact number of services in the original and repackaged versions. Furthermore, we found that in 7,428 app pairs (99.49%) the names of all of the services are identical. There are 9,914 original and repackaged app pairs that have at least one receiver component and that have the same number of receivers in the original and repackaged versions. Likewise, we found that in 9,857 app pairs (99.43%) the names of all of the receivers are identical. Similarly, we found that 481 app pairs out of 492 apps containing content providers (97.96%) have identical content provider names. These results are illustrated in Figure 4.18.

We used cosine similarity to compute the similarity measure. For each pair of original and repackaged apps, we compared the component names. For example, we compared the names of activities in the original app with the name of the activities in the repackaged app. Based on the definition of cosine similarity, we calculate the similarity by measuring the cosine of the angle between two vectors. For example, for activities, term frequency of the activity's name is the vector. Note that the permutation of activity names for each app will not alter the result. For example, the following two sets of activities' name have a cosine similarity equal to one:

"com.revmob.ads.fullscreen.FullscreenActivity|com.abarakat.webview.WebViewActivity|com.googl
e.ads.AdActivity|com.appbrain.AppBrainActivity", and

"com.revmob.ads.fullscreen.FullscreenActivity|com.google.ads.AdActivity|com.appbrain.AppBrainA
ctivity|com.abarakat.webview.WebViewActivity"

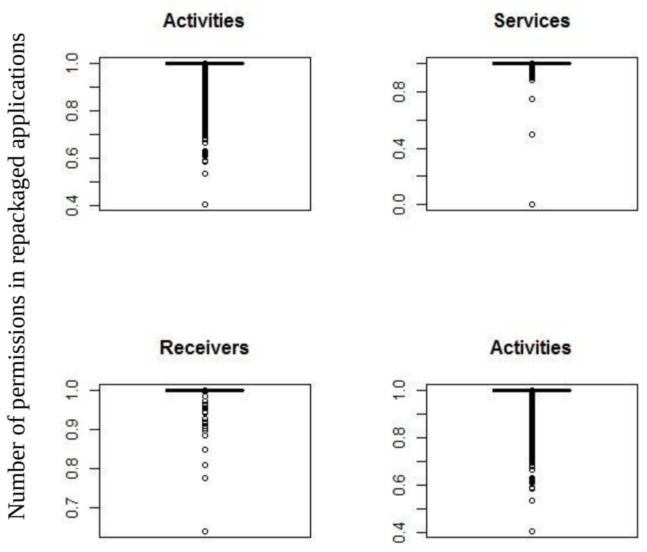


Figure 4.18 Boxplot showing the cosine similarity measure between the name of components in the original apps and the names of the components of their corresponding repackaged apps

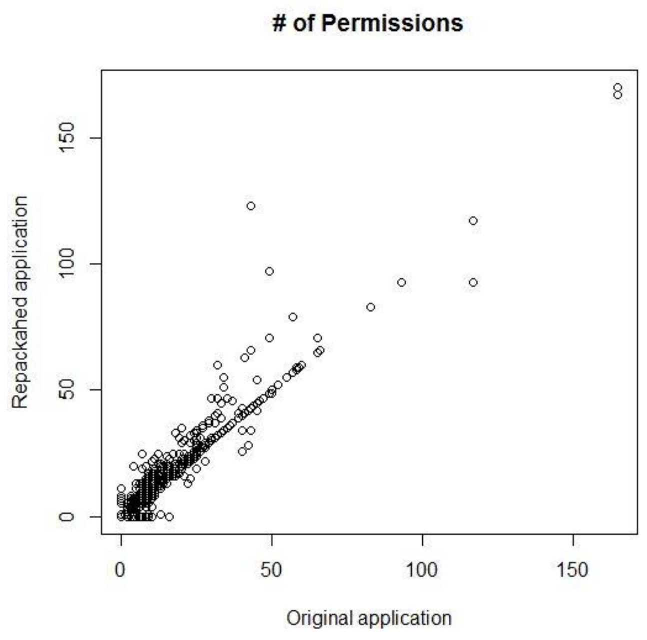


Figure 4.19 Number of permissions in original apps compared to the repackaged apps

F18: The number of permissions in repackaged apps is the same as that in the original apps in 93.34% of the cases. Figure 4.19 compares the number of permissions in repackaged apps with the number of permissions in their original pair. The figure shows that most of them have a similar number of permissions. Indeed, there are 14,109 pairs that have at least one permission and that also have the same number of permissions in the original and repackaged versions. In this group of 14,109 pairs, we find

14,045 pairs (99.55%) that have the same permissions in the original and in the corresponding repackaged version. This result is different from the result presented by Li et al. (2017b) where they found out that more permissions are added in most of repackaged apps. However, they also mentioned that there are some apps that some permissions added in repackaged one while such permissions exist in original one too. Note that the result we provide here is based on the distinct permissions exist in an app.

In the remaining 64 app pairs out of 14,109 pairs, however, some permissions were altered. We investigated the changes in the permissions requested in these cases. Figure 4.20 shows the permissions deleted from the original apps and Figure 4.21 shows the permissions added in repackaged versions. Two important conclusions must be stressed. First, the figures show that the deleted permissions are mostly specific to the app and start with the app's package name such as "AppPackageName.permission.C2D_Message". This permission relates to Android cloud device messaging (C2DM service) and allows the developer to push data to the app installed in the user's device from a server. Note that C2DM service was discontinued for existing apps and shut down completely in October 2015 (Google Inc. 2015) and that this permission was replaced with newer C2DM permission. Clearly, it connects the app to a new server. Second, in 5 samples, permission JPush_Message and CHECK_LICENSE was deleted. The permissions are needed if a remote server needs to connect an app and check the license of the app. This deletion suggests that malware developer blocked the connection of the original developer in the repackaged malware.

There are 663 pairs where the repackaged version has more permissions than in the original version. Figure 4.21 shows the frequency at which certain permissions were added. Since the permissions were added by malware developers, we can be certain that they are used to perform malicious operations. We provide this list of permissions to the public and to interested researchers working in malware detection. The top ten permissions added, in order of frequency, are the following:

android.permission.GET_TASKS,

android.permission.SYSTEM_ALERT_WINDOW,

android.permission.WAKE_LOCK,

android.permission.VIBRATE,

android.permission.READ_PHONE_STATE,

android.permission.ACCESS_NETWORK_STATE,

android.permission.WRITE_EXTERNAL_STORAGE,

android.permission.ACCESS_COARSE_LOCATION,

android.permission.INTERNET,

android.permission.ACCESS_WIFI_STATE.

4.8.4 Analysis, discussion and implication

In summary, our findings show that repackaged apps maintain almost the same number of components with the same names after repackaging. This feature can be used to detect repackaged apps. Traditional approaches for detecting repackaged apps rely on pairwise comparisons of apps to identify similar apps (e.g., (Zhou et al 2012a), (Shahriar and Clincy 2014), (Crussell et al. 2012), and (Crussell et al. 2015)). The main drawback of these approaches is their high time complexity since each app must be compared with all other apps. Taking advantage of the findings in RQ4, in the next subsection, we propose a novel app indexing scheme that relies upon the name of activities to cluster apps with similar activity names.

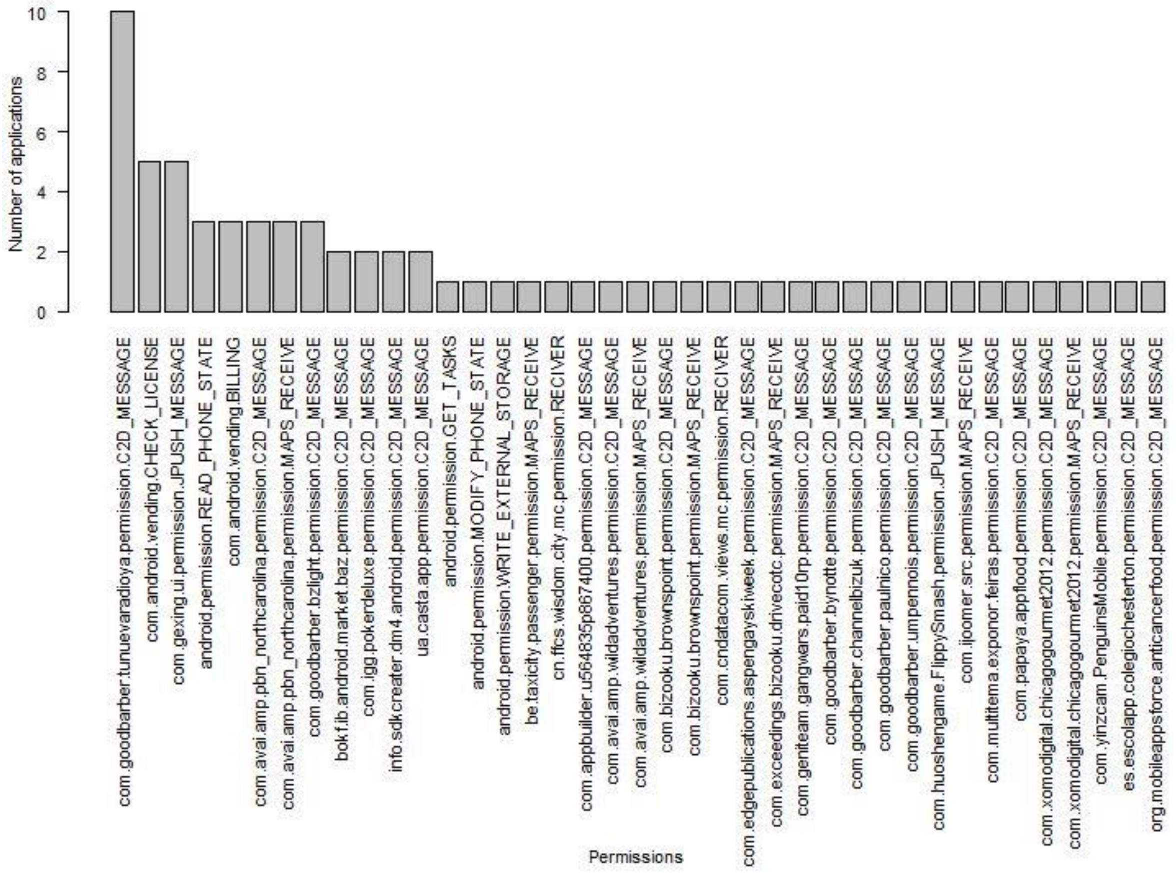


Figure 4.20 Permissions deleted in repackaged apps

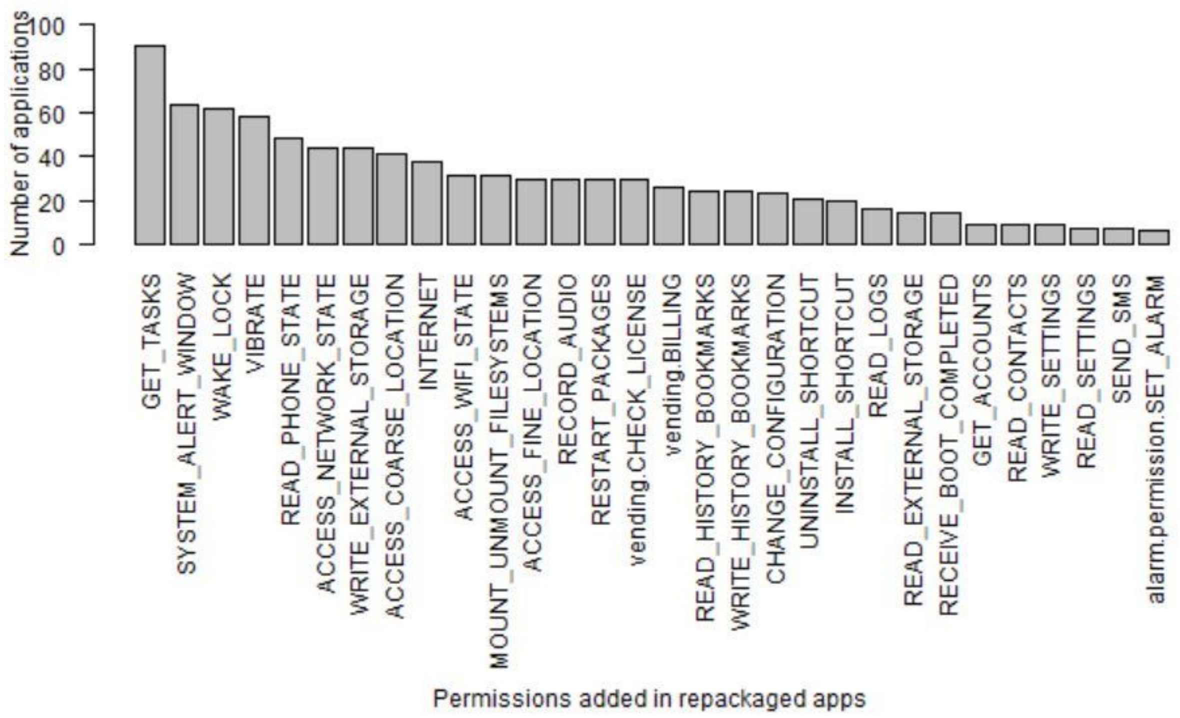


Figure 4.21 Permissions added in repackaged apps

4.9 Indexing Scheme

The aim of this scheme is to decrease the number of comparisons needed to find repackaged apps when using the indexing system. We used a piece-wise hashing strategy in which apps that have activities with almost similar names are assigned almost similar hash values. We used Ssdeep (Kornblum 2006) for hashing the name of activities. Then, following a method proposed by Winter et al. (2013), we used the n-gram of the hash as the look-up key referring to a row in our indexing table. We thus obtain a table where each row refers to an array that contains the ID numbers of apps and the row number is an n-gram generated by hashing the name of activities. Here, we used 7-gram because Winter et al. mentioned that, in Ssdeep similar hashes must have common 7-grams (Winter et al. 2013). Note that we have removed the activities coming from ad libraries since they are similar in a considerable number of apps and thus increase the number of comparisons though not providing much differentiation between apps. These activities are written in the ad packages. We obtained a list of ad packages from a study by Book et al. (2013). Table 4.6 shows the names of activities of a repackaged app (repackaged app1) and its original pair (original app1) and the way that they are used for indexing. As it is shown in Table 6, repackaged app1 with SHA 89EF7AC73CD35E588DAFC6646436BE586299EBDB246452E8D5E20B7233BBD1B4 is the repackaged version of the original app1 with SHA

AE064DED6254AD4803F55E441B871038AC7234CBEBCB7E6E24AAA466809438A2. Both of them have activities with the same name. Therefore, even after removing activities related to ad libraries, they have a similar hash, and they will be recorded in the same row of the index.

Another original app has SHA D7546FB04BC9ABB901017848BC49251AF038DC7E17E013523E5567AB6FFE0BC5. In what follows, we refer to it as original app2, and it is not the original version of repackaged app1. Some of the activities of this app share a similar name to the activities in repackaged app1. Therefore, by using n-gram of a hash of activities name for indexing, repackaged app1 and original app2 will be also indexed in the same row as well.

Table 4.6 An example of the name of activities used for indexing

Item	Original app1 with SHA: AE064DED6254AD4803F55E441B871038AC7234CBEBCB7E6E24AAA466809438A2 *	Original app2 with SHA: D7546FB04BC9ABB901017848BC49251AF038DC7E17E013523E5567AB6FFE0BC5
Activities' name separated by delimiter "|"	com.revmob.ads.fullscreen.FullscreenActivity|com.abarakat.webview.WebViewActivity|com.google.ads.AdActivity|com.appbrain.AppBrainActivity	com.revmob.ads.fullscreen.FullscreenActivity|com.abarakat.webview.WebViewActivity|com.google.android.gms.ads.AdActivity|com.appbrain.AppBrainActivity|com.bqquqi.cdueey177143.MainActivity|com.bqquqi.cdueey177143.BrowserActivity|com.bqquqi.cdueey177143.VDActivity
Activities after removing ads	com.abarakat.webview.WebViewActivity	com.abarakat.webview.WebViewActivity|com.bqquqi.cdueey177143.MainActivity|com.bqquqi.cdueey177143.BrowserActivity|com.bqquqi.cdueey177143.VDActivity
Hash of activities	QuEXnfA/AHzMAX3Qcn	QuEXnfA/AHzMAX3QcS6LXOSIMyQGRZcS6LXOSIMZH/qARcS6LXOSIMfRcn
n-grams	QuEXnfA, uEXnfA/,EXnfA/A, XnfA/AH,…	QuEXnfA, uEXnfA/,EXnfA/A, XnfA/AH,…

*Repackaged app1 with SHA 89EF7AC73CD35E588DAFC6646436BE586299EBDB246452E8D5E20B7233BBD1B4 has activities with the same name as Original app1 with SHA AE064DED6254AD4803F55E441B871038AC7234CBEBCB7E6E24AAA466809438A2

We computed the Ssdeep hash for all original and repackaged apps in our dataset. The hash is in Base64 encoding and each digit in Base64. Identifying the best piece-wise hash functions or finding the best indexing approach and evaluating the performance of the scheme is beyond the scope of this thesis. Here, we seek to show that using the names of activities for indexing can decrease the number of comparisons needed for pairwise comparison of apps. In traditional pairwise comparisons each repackaged app (15,976 apps in our dataset) must be compared with all original apps (2,776 apps in our dataset). We calculated the number of pairwise app comparisons needed when using our new indexing system. The result is given in Figure 4.22. The median is 12; which means that on average, each app must be compared with 12 other apps. It may also happen that two apps are indexed together in more than one row. When this occurs, we compare the two apps only once.

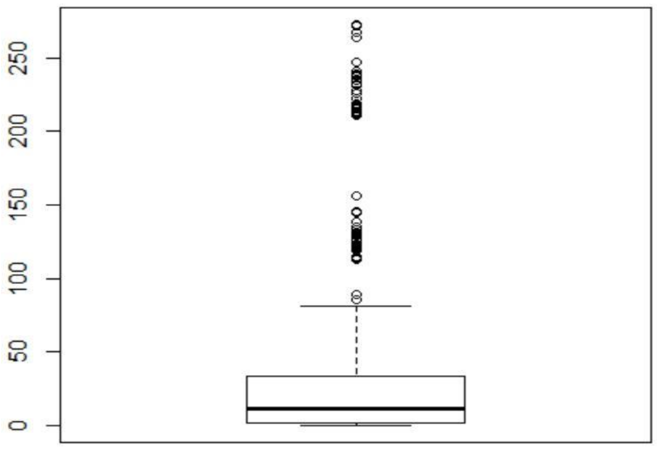


Figure 4.22 Number of comparisons for each repackaged app needed in the proposed indexing system

4.10 Threats to Validity

The selection of the dataset is one of the most common threats to validity for empirical studies. It is possible that the selected apps share common properties that we are not aware of and therefore, invalidate our results. We mitigated this threat by using AndroZoo, which was developed by researchers to advance the field of mobile security. The dataset was carefully built to contain Android apps from various categories. We also used Kaggle, which is a very rich and diverse source of Android apps.

Another threat to the generality of our study relates to the kind of repackaged apps present in the AndroZoo dataset. All of the repackaged apps in AndroZoo are detected as malware by at least one anti-virus. Therefore, our study is limited to repackaged apps containing malware. However, there may be repackaged apps where no malware is detected. Li et al. (2017b) categorized repackaged apps and named the ones having malware as *piggybacked apps*, and this is the only type of app examined as part of this study. In fact, our study includes only piggybacked apps. Obtaining a dataset of repackaged apps which do not contain malware and study them is left for future work.

Another threat is the drawback of tools used in our study such as APKtool (Wisniewski 2012) and Dex2jar which may vitiate the accuracy of results. We mitigated this threat by using findings based on statistics over a substantial number of apps.

To identify files in a repackaged app that are manipulations of files in its corresponding original app, files that are added in a repackaged app, and those of the original app that were deleted in the repackaged apps, we used two thresholds t1=70% and t2=98%. We identified these values through experimentation. Different values of t1 and t2 may impact the results. To mitigate this threat, we checked the files of a large number of repackaged-original app pairs to validate the results.

There is also the threat of time inconsistency where the time duration between the date of recorded meta data and release of apps are different for all apps. To mitigate this threat, we grouped the apps according to their release date in finding F7 and finding F8. However, the release date of the apps in the AndroZoo dataset may not be accurate, as was shown in a study done by Li et al. (2018) about the release time inconsistency or shown in a study by Salem et al. (2019) about the malware report. Indeed, they found that in 48% of app pairs in AndroZoo, the repackaged apps release time is anterior to that of the original apps.

Another threat to validity concerns our findings related to the API calls added and deleted during the repackaging process. As explained in RQ2, when the name changing obfuscation is present in a repackaged app, the API called identified as deleted or added may be the same. Nonetheless, we may detect these APIs as added\deleted because the obfuscated files have a low similarity with the original app. This fact does introduce a small possibility of error. In our dataset, this scenario is actually rather uncommon since less than 4% of the repackaged apps exhibit obfuscation of a type “name changing” that would introduce this error.

A final threat to validity concerns the comparison between the apps in our dataset and other apps in the stores that are not repackaged (see finding F7, F8, and F9). We obtained information about these apps from Kaggle dataset and from the study performed by Dong et al. (2018b). The threat to validity is that we cannot be sure if those apps are repackaged version of any other app or not. We can only say that we are comparing the repackaged app with the rest of the apps in the world. This threat would exist even if we downloaded apps randomly from stores as well.

4.11 Discussion and Conclusion

Summary. We have performed an empirical study comparing repackaged apps and their original versions. We used the dataset available on AndroZoo (Li et al. 2017a) containing 2,776 original apps and 15,296 repackaged app. Each original app is associated with a multiplicity of repackaged apps. In total, our dataset contained 15,296 pairs of original and repackaged apps. Our study sought to answer five research questions. The first question is: what are the main motivations for repackaging. We found that statistically, the main motivation for repacking is the distribution of adware. The second question examines how the apps' code is manipulated during the repackaging process. We found out that in repackaged apps, adware use API calls that are similar to those that occur in legitimate apps containing advertisement. In the third research question, we examined which types of apps are most frequently targeted for repackaging. Our finding showed that apart from the popularity of apps, the absence of obfuscation is a significant factor in determining if an app will be repackaged. In the fourth question, we asked why users download the repackaged apps even though the original one is freely available. We first showed that repackaged apps often have a high popularity and are located in trusted app stores. Then, an analysis of the name of apps suggested that users might erroneously believe that they are downloading a newer release of the original app or the local version of it in a foreign language. The last question studied the attributes of repackaged apps such as apps' components and permissions. Our findings show that in repackaged apps, these attributes remain similar to those of the original apps, suggesting that those attributes cannot be alone serve as the identifying features for detecting malware. However, they can form the basis of an indexing system used to find similar (and possibly repackaged) apps. Note that the detection of repackaged apps is different from malware detection. In the first case the object is to look for similarities between apps while in the latter case, the goal is to detect malicious behaviors.

Suggestion to protect apps against repackaging. Based on the findings in RQ3, most of the apps that were repackaged do not exhibit obfuscation. This suggests that malware developers need to understand the code of the app that they manipulate. Therefore, to protect apps against being repackaged we suggest applying obfuscation the app, which can hinder comprehension of the code.

Suggestion for malware detection. As discussed in RQ2, the frequency of API calls does not differ much between original apps and repackaged apps containing adware. Therefore, a detection mechanism for adware based exclusively on API calls is unlikely to be successful. Since, as it is shown in RQ1, a large proportion of repackaged app contains adware, as opposed to other types of malware, we suggest that all detection approaches investigate the accuracy of their approach using a dataset that contains apps repackaged with adware as well as the corresponding original versions of the same app.

Validity of common assumptions. Previous studies, e.g. (Li et al. 2017b), have shown that more permissions are requested in repackaged apps in comparison to the original apps. However, our study shows that 93.34% of repackaged apps have similar permission as their original counterpart. Therefore, using permission as a feature for detecting malware is not practical. Li et al. (2017b) mentioned that they found that repackaged apps sometimes requested the same permission multiple times, when it was already requested by the original app. Here, we did not distinguish between individual and duplicate requests for permissions.

In investigating RQ4, we found that a considerable number of repackaged apps do not have a name that is very similar to the name of the original app. For example, an app named "Pregnancy Exercise" was repackaged with names such as "Dance Waltz" and "Cha Cha Cha". We also found out that many repackaged apps are republished in a different language than the original app. Some repackaged app detection methods, such as the ones proposed by Sun at al. (2015) or Soh et al. (2015), are premised on the fact that malware developers endeavor to maintain the "feel" and "see" of the original app, to better dupe users into downloading their repackaged apps. We have found this assumption to be erroneous in a considerable number of apps. If there are similarities in the appearance of apps, it is simply because the malware developers do not put much effort alter it. In other word, keeping the same "feel" and "see" is not necessary. However, further study is needed to determine if the apps' resources, such as images, are kept similar in the repackaged apps. Still, based on our findings, detection approaches, which are based on this fact should be adopted only with caution. Overall, we can also say that attackers seem prefer popular apps.

Future work. Studying the resource manipulations performed in repackaged app could yield a promising strategy to detect repackaged apps. Furthermore, since one of the motivation of users to download repackaged apps seem to be to obtain a legitimate apps in a language in which it is unavailable, the choice of the language used in repackaged app's user interface is important aspect of future investigation.

The introduction of adware usually leads to the insertion of new code in the repackaged version, as well as to the deletion of other segments of the original code. Based on our findings in RQ2, the APIs used in the added parts are very similar to the one present in deleted code; often serve to connect to a server. This suggests that any approach that can make it difficult for adware developers to identify those parts of code such as code obfuscation techniques could be a promising defense mechanism against repackaging.

Finally, we provide a characterization of the differences between repackaged and benign apps. The next step is to dig deeper and examine other properties such as code quality metrics, the use of libraries, process metrics (developer’s experience, etc.) to further explore the problem by answering the question of why the repackaged payloads are introduced.

Chapter 5 An Exploratory Study of the Service Lifecycle of Android Apps

5.1 Introduction

Since repackaging is a popular way to broadcast malware, in this chapter, we investigate how malicious code can be embedded into legitimate apps. More precisely, we focus on the services and notifications that malicious apps provide as compared to legitimate apps. The use of services stems from the fact that malware must continue operating even after the original app terminates. In addition, malicious apps need to provide users with the same experience as when using normal apps. Therefore, any malicious operation has to be designed in a way that goes undetected by users.

Some studies indicate that malicious operations are located in service components. Aafer et al. (2013) show that the API calls related to starting or stopping a service component, such as *android.app.Service.onDestroy()*, *android.app.Service.onStart()* and *android.app.Service.onCreate()* are among the top twenty most frequent API calls found in malware samples. Additionally, Xu et al. (2016) showed that around 30% of malware declare intent filters (types for message objects) for services while only 7% of benign apps do so. Therefore, we focus on services to model the behavior of apps and to distinguish malicious and benign operations. In fact, although different benign applications can have varied functionality and behaviors, services in benign apps tend to exhibit uniform behavior patterns. Thus, in this study, we draw upon differences in usage patterns of services to distinguishing malware from benign apps.

To investigate our assertion, in this chapter, we study the lifecycle of Android apps' services. We define the service lifecycle based on the execution flow of a service in an Android system through calling callback methods. Moreover, we study the service execution from the view point of the app's user. The aim is to extract a number of features that differentiate the malicious and benign services.

We perform experiment over the dataset of benign apps downloaded from Google Play Store and malicious services found in Genome malware dataset (Zhou and Jiang 2012b) and AndroZoo repackaged malware dataset (Li et al. 2017). We provide systematic approaches to finding malicious services in

Genome dataset and AndroZoo dataset. Then, we study the code of the services to extract the features, which differentiate malicious and benign services.

The remainder of this chapter is organized as follows. Section 5.2 covers an overview of service's lifecycle. Section 5.3 presents the experiment performed on malicious services in Genome dataset and benign services in Google Play Store. Section 5.4 discusses the study over malicious services on AndroZoo dataset. Section 5.5 examines the threats to validity. Finally, the conclusion and Future work is discussed in Section 5.6.

5.2 Overview of lifecycle of apps' services

As explained in chapter 2, a typical Android app contains four types of components: Activities, Services, Content providers and Broadcast receivers. Activities represent what the user can do with the app. A service is a component that runs in the background. A content provider manages a shared set of the app's data. A broadcast receiver is a component which responds to system-wide broadcast announcements such as "the battery is low". Activities, services and broadcast receivers are activated by an asynchronous message object called an intent (Android Developer Documentation 2018).

From the perspective of the user who runs an app, the app's components can be divided into two parts. The first part consists of the foreground part, which contains activities that a user can use. Activities are, in fact, components that should provide the same experience to the user in both the repackaged and the original app. The second part of the app is the background components including services, broadcast receivers and content providers operating in the background and the operation's result is sent to the user in activities. By this categorization, if a malware wants to operate for a long time, hidden from the user, it is probable to be added to background components. This is the reason this study focuses on analyzing the lifecycle of services. As we explained in Section 2.3, some callback methods are called through a service lifecycle.

In AndroidManifest.xml, a service is defined by a <service> tag. It has attributes such as name, *exported* attributes that show if other applications can use this service, and an intent filter, which lets the Android system respond to intents. The service is implemented in Java as the subclass of the class `Service`. The class name has to be similar to the name defined in AndroidManifest.xml for the service. A service in Android can started in two forms, *started* or *bound*. In the started form, a service is started by an another component of the app by calling method `startService()`. It will continue to run indefinitely in the background even if the user switches to other applications. It will be stopped by calling `stopSelf()` or `stopService()` methods. In the bound form, a component is bound to a service that it uses by calling the method bindService(). A bound service offers client-server interface, allowing the components to interact with each other. A bound service is destroyed if all application components is unbound by calling method unbindService(). Usually, for long running operations and single tasks, the startService() will be used.

Based on the nature of usage for these two kinds of service forms, started and bound, if a stealthy operation wants to be added to an app, it can be embedded in service and started when the startService() method of this service is called. Thus, it will have a chance to run indefinitely in a device. Since it is the user who decides which activity should operate while running an app, the bound service, whose public methods are called by the activity, is indirectly controlled by the user. However, when an activity runs a service, the service can continue running even if the user quits running the app.

When the implemented task for a service is finished, the service usually notifies the user of its completion. Notifying a user of a service operation can be performed using any one of the following three methods:

- By generating a notification object and passing it to the system. Notifications may contain actions, which can start an activity.
- By starting an activity since activities are the foregrounded components and users can see the results of the task done in services.
- Through message passing by having the component give permission to the service to send a message by passing message-handler to the service while starting it. Some protocols such as AIDL are designed to let other applications use the service of other applications. They are also based on the message handling approach.

The Android system tries to keep an application process alive as long as possible, but sometimes it needs to kill some processes to reclaim memory for new processes. It terminates processes based on their level of importance to the user. For example, the processes containing activities the user interacts with are given highest priority. It should be noted that services that are terminated may be restarted as standalone with a starting form, instead of a bound form.

Considering the event-driven nature of Android applications, we can group the operations performed in a service into two categories. The first one contains all the operations that can be started after the service is created by an activity. It will continue to operate, and halt after completing its operation. The user may be notified of the finished task. The second category contains tasks that are executed in special situations. For example, the service is started when a broad cast receiver starts it after a particular event happens such as the event of receiving messages or emails. In fact, it is the system that indirectly started the service.

A broadcast receiver is defined through the <receiver> tag in AndroidManifest.xml of the application. It also contains an attribute, set to “true” if it can receive messages from sources outside its application, or to “false” otherwise. The default value depends on whether the broadcast receiver contains intent filters. The absence of any filters means that it can be invoked only by intent objects that specify its exact class name. In this case the default value is "false". On the other hand, the presence of at least one filter implies that the broadcast receiver is intended to receive intents broadcasted by the system or other applications, and in that case, the default value is “true".

When a receiver is defined by an intent filter, it allows the other app or components of the app send intents to this receiver. An app also can define the intent filter in the receiver to obtain the implicit intents from the system, for example, getting alarms when the system is rebooted. In the next section, we will examine this feature to see if malicious apps behave in such a way, or not. Note that a broadcast receiver can only be started; it cannot be bound to a service.

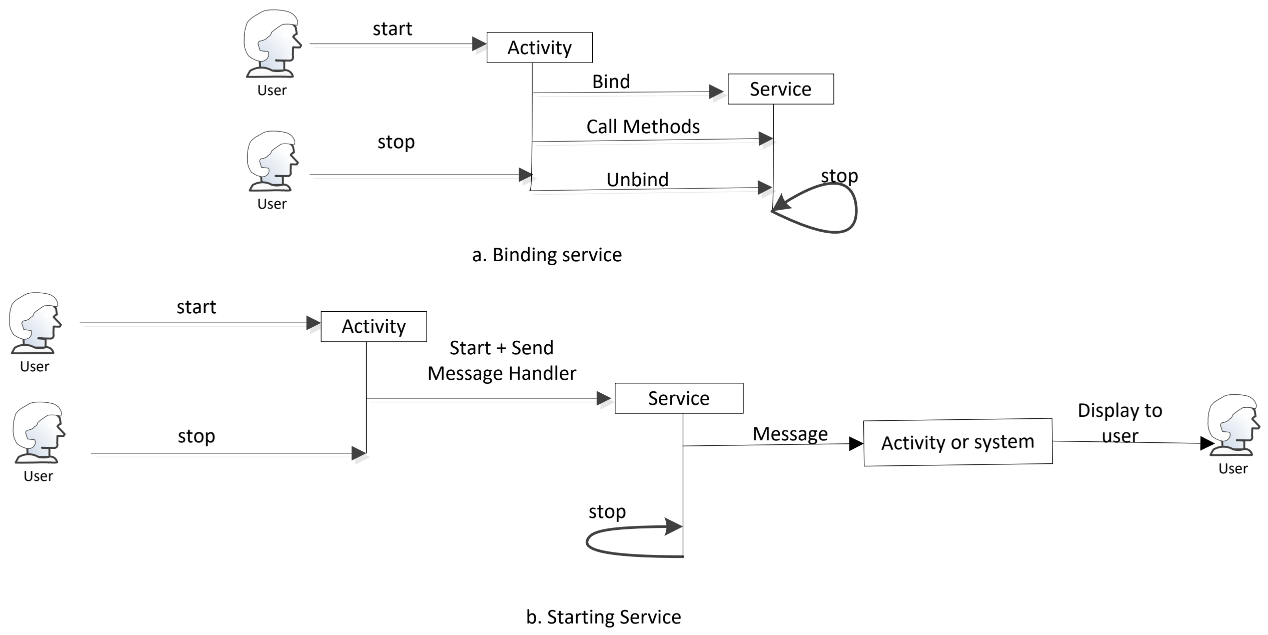


Figure 5.1 Android app's service lifecycle

Figure 5.1 illustrates the communication of a service during its lifecycle. This figure shows how a user as an actor in the lifecycle of components can start a service and be notified from the service operation. It should be noted that in this figure, instead of activity, there can be any other type of app’s component. As shown in Figure 5.1, the user’s role is important for this study since malware that attempts to hide itself, would operate stealthy by avoiding any communication with the user. A suspicious behavior may motivate users to use expert tools to detect and remove malwares. In addition, malware that perform malicious operations involving user interactions such as sending messages to user contacts or alarm the user by erasing files, are usually not published on reliable app stores such as Google Play Store (Tam et al. 2015). Rather, they are most found on third party (untrusted) app stores.

With the focus on the services of an app, we can define each app by the set of services it has. Each service consists of some features that characterize the app's behavior. The defined features are summarized as follows:

- The service permissions used by the service; a service that does not need any permission does not perform any malicious operation.
- The notifications generated by a service to show if it has interactions with the user; this can be done by generating notifications or sending messages to any message handler, which is passed to a service while starting it.
- The list of activities which start the service or are bound to the service; if a service is bound to a component, it has access to its public method. Thus, it is indirectly connected to other components and will stop as soon as other components unbind it.
- The list of activities started directly by the service; as activities are visible to users, it can be a way of notifying the user of the operation of a service.
- List of broadcast receivers in the app that start a service; these broadcast receivers are defined by their intent filters. They can help determine when a service starts to perform actions. Note that a broadcast receiver cannot be bound to a service.

The list of services is easily extracted from AndroidManifest.xml file. The tag <uses-permissions> in AndroidManfest.xml shows that permissions are needed by the app. These tags are usually defined independently from the components, making it difficult to know which components use which permissions. Thus, we need to study the application's code and APIs used in each class to identify the permissions used by a service. In the AndroidManifest.xml, for each component, android:name attribute specifies the class name of the component in the corresponding Java code.

We used a tool called Androguard[4] to extract the methods called in a class and the permissions used by them. We wrote a python script[5] over Androguard that recursively follows the methods called in a service class, starting from callback method onStartCommand() and onBind(). All the permissions used in these methods specify the permissions of the service.

[4]https://code.google.com/p/androguard/
[5] https://github.com/kkhanmohammadi/codes

We were also curious to know if a service notifies the user of its operations. As such, we extracted all the notification objects generated in the methods called during the service lifecycle. Apart from that, we also extracted direct call for activities. As activities run in the foreground, they inform the user of the operations done in service. As discussed in the previous section, our assertion is that stealthy services don't do this. In this regard, we also extract the messages passed by the service to other components. The tool Androguard let us find the object Notification and Message in a service. The other features we extracted were the broadcast receivers and the activities, which call the service as well as activities called by the service. We used Androguard to extract the callback methods called in activities and broadcast receivers. Thus, we found if a service is called on those components through callback methods. When a service is started by a broadcast receiver, it shows the special situation in the device that a service called, such as "a message received" or "battery is low".

5.3 Study on Genome malware dataset and Google Play Store benign apps

5.3.1 Dataset

We used the Genome malware dataset for our study. We first needed to extract the malicious services that would be used for classification. We obtained the names of the services in each app from the Androidmanifest.xml file by using Androguard (Desnos 2015). The Genome repository contains 1226 malware apps categorized in 49 families of malware. This repository includes malwares dating back from 2012. Based on code similarity of services in a variety of samples in each of the malware families, we identified the malicious services in each malware family. We studied the code of such services to confirm the existence of malicious operation. In this regard, we retrieved the source code from the samples by using dex2jar[6] which converts a dex file into a Java source file. To increase our confidence in the results, we also studied the malware analysis reports provided by Zhou and Jiang (2012b). In sum, we found 67 malicious services in the malware apps of the Genome dataset.

The legitimate apps were downloaded from Google Play Store in April 2015. We downloaded 200 apps randomly from the first twenty categories in Goggle Play Store; ten apps from each category. Note that there is a threat to validity where we assumed that apps in Google Play Store are benign.

5.3.2 Analysis approach

As we explained, malware try to hide their operations by running in background services and having no communication with the user. We can classify the feature of the services as follows. The first feature,

[6] https://github.com/pxb1988/dex2jar/wiki.

PERMISSION, represents the permissions used during the service life span. The second feature, MESSAGE, is the notifications or messages passed to activities or shown to the user. The list of permission needed by an app is written in the Androidmanifest.xml of the app but not all of those permission may use by a service component. Therefore, we used the following approach to determine which permissions is used by a service. We first used Soot library (Bartel et al. 2012) to extract API calls through the lifecycle of a service. We obtained the call graph of a service component, where the callback methods are the entry points and extract the APIs called in the methods in such a graph. Then, we used the output of Pscout (Au et al. 2012). Pscout, studied the code of Android APIs and provide the list of APIs needed for specific permissions. We examined the APIs called in a service lifecycle to understand if any of them needs permission to execute. Therefore, we obtained the list of permissions needed by a service. We also searched the APIs in a service lifecycle for the Notification API.

We also extracted two other features. The first one, CALL_ACTIVITY, shows if the service calls other activities after finishing its operation informing the user when the operation is done. The second one, SERVICE_BOUND, depicts if the service is bound because the bound services are alive as long as the component which binds them is alive. All of these features are also obtained by our code that searches the APIs called in the service call graph.

We extracted the features for all legitimate apps downloaded from Google play store and the malware dataset. The results are shown in Table 5.1. We use the following labels to refer to groups of the various features of services:

A: NO PERMISSION

B: PERMISSION and MESSAGE

C: PERMISSION and NO MESSAGE and NO CALL_ACTIVITY and NO SERVICE_BOUND

D: PERMISSION and NO MESSAGE and (CALL_ACTIVITY or SERVICE_BOUND)

E: NO SERVICE

F: NO IMPLIMENTATION

Note that in this table, having “NO” before the feature’s name means that the app contains at least one service that does not have this feature. For example, NO MESSAGE means that the app contains a service which does not send messages or notifications.

While studying the apps’ services, we have found that some of the apps do not have any services and as such, we categorized them as NO SERVICE. There were some apps that contain only the definition of a service in AndroidManifest.xml, but the service is not implemented in the source code, and so they have been categorized as NO IMPLEMENTATION. This may be due to the developer’s mistake in keeping unnecessary service definitions in AndroidManifest.XML or the service’s code may be loaded at run time.

It should be noted that if there is an app containing services with different features, we take into account the most restrictive services. For example, if a normal app contains two services where the first one is PERMISSION and MESSAGE and the second one is PERMISSION and NO MESSAGE and CALL_ACTIVITY, we categorize it in the third group in Table 5.1.

Table 5.1 Results of classifying apps based on their services

Category	**A:** NO PERMISSION	**B:** PERMISSION and MESSAGE	**C:** PERMISSION and NO MESSAGE and NO CALL_ACTIVITY and NO SERVICE_BOUND	**D:** PERMISSION and NO MESSAGE and (CALL_ACTIVITY or SERVICE_BOUND	**E:** NO SERVICE	**F:** NO IMPLIMENTATION
Normal (200)	25 (12%)	58 (29%)	14(7%)	76 (38%)	19 (9.5%)	8 (4%)
Malwares (65)	1 (1.5%)	3 (5%)	39(60%)	14(21%)	8 (12%)	0 (0%)

5.3.3 Results

Apps that contain services with NO PERMISSION show that they have safe services. Clearly, a service performing malicious operation needs permissions. As it is shown in Table 5.1, 25 out of 200 (12%) of the apps from the legitimate apps dataset fall in this category while only 1 out of 65 (1.5%) of the apps in the malicious dataset do so..

The number of apps with PERMISSION, NO MESSAGE, CALL_ACTIVITY shows a significant difference between the apps in the malware dataset and those in the legitimate app dataset.

The number of apps in Group D which contains the services that are bound to a component and will stop after the component unbinds it; and in group B, which contains the services with MESSAGE are significantly larger than that in normal apps. On the other hand, the number of apps in group C, which contains the apps with services that have no connection with rest of the app after being started, are larger than other groups in the malware dataset. These results show that malwares and normal apps have different behavior with respect to the service lifecycle and its connections and communication with the rest of the app.

In order to gain a better understanding of malicious apps that have NO SERVICE or NO PERMISSION, we further examined sample malicious apps. We use the code of malware and information provided by Zhou and Jiang (2012b) and by Felt et al. (2011).

Among the group of malwares that do not have services, FakeNetFlix and FakePlayer ask for user credential information directly from the user via an activity component. This sort of malwares is easily detectable by a security expert. DroidDeluxe and some version of Asroot do not contain malicious payload on the app itself. They obtain root privilege by exploiting a vulnerability during installation. Similarly, DroidKungFuUpdate, AnServerBot, BaseBridge and Plankton get root privilege to download and install malicious apps. SMSReplicator, Walkinwat, YZHC do not have services and perform the malicious operation when an event is received by a broadcast receiver. For example, SMSReplicator has a broadcast receiver that listens to incoming messages and forwards them to a selected number.

AnserverBot is a malware that asks users for update and takes that opportunity to install the malicious payload. Therefore, the services in the malware itself do not require any permissions.

ADAR, which has PERMISSION and MESSAGE, use media player to send notifications. One version of Asroot and BaseBridge also uses notifications while updating for malicious payload. SNDApps notification is sent by a service in the repackaged app that the malicious payload added to the original app. It was the only sample in the malware dataset that adds the malicious operation to the existing service of a legitimate app. However, there was another service related to malware in particular. This shows that studying the content of messages in a service can help detect suspicious services.

In summary, the results show that:

- 68% of malwares that contain services require permissions and do not communicate with rest of the components in the apps after being started.
- 92% of the studied legitimate apps notify the user of the app about their operation of background services by sending notifications, by passing messages or by activating visible components in the app.
- Services in nearly all the malware, and in more than 80% of the benign apps use permissions requested by the app.
- The apps that upload malicious code through updating do not have services.
- All the services in malware are started and not bound by other components of the apps.

5.4 Study on AndroZoo malware dataset

5.4.1 Dataset

In this section, we study the AndroZoo dataset (Li et al 2017a), which contains a list of current malware. The AndroZoo dataset contains 15, 296 pairs of original and repackaged malware apps. This dataset forms the basis of our study, in which we endeavored to compare the services in repackaged malware with those in the corresponding original app.

5.4.2 Analysis approach

7, 689 out of 15, 296 malware samples have services. We obtained a list of the services in each app by studying the AndroidManifest file of each application. We used Androguard (Desnos 2015) to extract the names of services in the AndroidManifest.xml file. In our study, we aim to identify which services from the repackaged app contain malicious operations. We first extracted a list of samples in which one or more services have been added in the repackaged version but are not present in the original pair. In a second step, we used the tool, called Euphony, provided by (Hurier et al 2017). Euphony use text mining approaches to study the VirudTotal report of each malware to find the name and type of the malware. Therefore, we used this tool to identify the name and type of malware, and grouped the services accordingly. Finally, within each group of malware, we identified the services, which are repeatedly added to the samples of malware in each group. We extracted the Java code of these services and studied their behavior. We used the tool "Dex2Jar" to extract the Java code of the apps.

We found multiple service components that are repeated in each group, but with their components altered. For example, in samples of malware named *Kuguo*, a service is named *com.android.mks.Ssreb* in one sample and is named *com.zyyj.cl.Ssrea* in another sample. The names of service components are different, but they perform similar operations. They may also have different names for identifiers (e.g. method and variable names).

Based on the study of service components, we classified services as either being malicious or benign services. Not every added service is malicious. For example, in a malware named *pircob,* the service *chatsalas.es.shoutcast.StreamService* is added in the repackaged app, but it does not perform any malicious operations, and it seems to have been added to the repackaged app only because it was included in a library required by the malware developer.

Table 5.2 provides a list of malware families that contain malicious services. Note that there may be additional malware samples in AndroZoo, which contain malicious services, but went undetected. Here we list the malware we obtained from applying the method detailed above. The table shows the number of samples that were found to contain malware with a similar name. It also contains types of malware, as identified by Euphony, and number of repackaged malware found with the same malware name and type (column named "Number of Samples"). The other columns in the table are explained below.

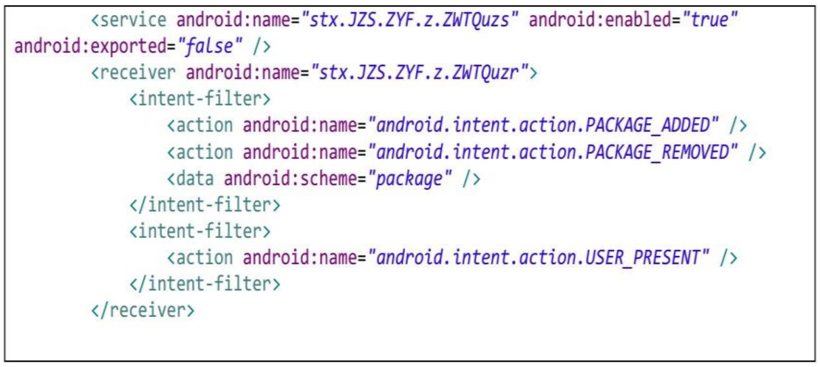
Table 5.2 List of Malware having malicious services in the Androzoo dataset

Name	Type	Number of Samples	Reflection	Name Change	Dynamic Loading	OnBind	Activity Call	Start Service
adpush	adware	1	No	yes	No	No	yes	onCreate of an activity
adwhirlads	adware	1	No	yes	No	No	yes	onCreate of an activity
adwo	adware	3	Yes	No	No	No	yes	onCreate of an activity
airpush	adware	12	No	No	No	No	yes	Broadcast Receiver
apkq	trojan	1	No	yes	No	No	No	onCreate of an activity
basebridge	trojan	1	No	yes	No	No	No	Broadcast Receiver
commplat	addisplay /adware	4	Yes	yes	No	No	No	onResume of an activity
cxqisl	trojan	1	yes	yes	yes	No	No	Broadcast Receiver
dcsfjy	trojan	2	No	No	No	No	yes	Broadcast Receiver
dianjin	adware	1	No	yes	No	No	No	Broadcast Receiver
domob	adware	3	Yes	No	No	No	yes	onCreate of an activity
dowgin	adware/ riskware/ spyware/ trojan	81	yes	yes	yes	yes	No	Broadcast Receiver
droidkungfu	backdoor /trojan	2	yes	yes	No	No	No	onCreate() of Main Activity
gingermaster	adware /trojan	2	yes	yes	yes	No	No	onResume of an activity
ginmaster	trojan	3	yes	yes	yes	No	No	onResume of an activity
kuguo	adware /trojan	11	No	yes	No	No	yes	onclick of activity
millennial mediaads	adware	1	No	No	No	No	yes	Broadcast Receiver
mtk	trojan	1	yes	yes	yes	yes	No	Broadcast Receiver
shixot	adware /trojan	2	Yes	yes	yes	No	No	onResume of an activity
startapp	adware	1	Yes	No	No	No	yes	onCreate() of Activity
umeng	addisplay	1	No	No	No	No	No	start by broad cast reserver
waps	adware	2	Yes	No	No	No	yes	start onCreate of an activity
wiyun	adware	1	Yes	yes	yes	No	No	broadcast receiver
youmi	adware	3	Yes	No	No	No	yes	start onCreate of an activity

5.4.3 Results

The followings are the main findings we obtained through our study of the code of the malicious services. We illustrate our findings by presenting the code of malware from the AndroZoo dataset.

F1: malicious services have only a loose connection with rest of the apps' code. The services in every sample that we studied are started by a broadcast receiver or an activity. In our samples (e.g., Basebridge and Dowgin), broadcast receivers usually wait for an action such as a system boot or the addition of a package to the device. Figure 5.2 shows part of the AndroidManifest.xml file of malware *Wiyun*. Here we see the definition of a service and of a receiver which starts that service. The actions are written in the intent-filter of the receiver in AndroidManifest.xml file. The action is caught by the callback method *onReceive()* upon which the receiver starts to execute its operations. The receiver then goes on to start the service.

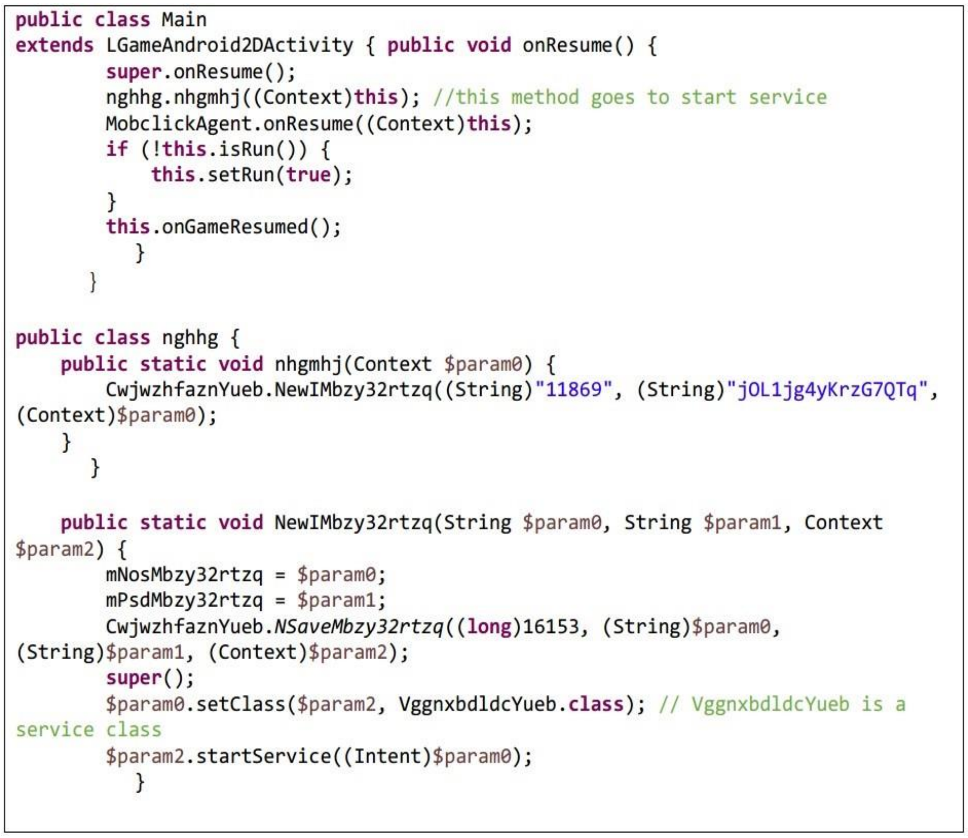


```
        <service android:name="stx.JZS.ZYF.z.ZWTQuzs" android:enabled="true"
android:exported="false" />
        <receiver android:name="stx.JZS.ZYF.z.ZWTQuzr">
            <intent-filter>
                <action android:name="android.intent.action.PACKAGE_ADDED" />
                <action android:name="android.intent.action.PACKAGE_REMOVED" />
                <data android:scheme="package" />
            </intent-filter>
            <intent-filter>
                <action android:name="android.intent.action.USER_PRESENT" />
            </intent-filter>
        </receiver>
```

Figure 5.2 Part of Androidmanifest.xml file of malware "Wiyun"

In some samples, the service is started by an activity. In every sample we obtained from the AndroZoo dataset in which the service is started in a callback method of the activity, no action is required to be performed by the user before the service is launched, nor is the launch of the service dependant on the satisfaction of a logical condition (such as one captured by an 'if' construct). Rather, the callback method starts the service in its main code path. Since callback methods are called by the system even when other parts of the app's code do not specifically call them, it is possible that services are started by the application without the intent of the application's user. In our dataset, two particular callback methods seems to be principally targeted by malware developers, namely *onCreate()* and *onResume()*.

Furthermore, the API needed to start a service is not directly called in the callback methods. Instead, we found that the callback method usually calls other methods, which in turn proceed to start the service. Figure 5.3 shows part of the code of an activity that starts the service. The code is written in the onResume() callback method. Table 1 includes a column showing how the services is started.



```
public class Main
extends LGameAndroid2DActivity { public void onResume() {
        super.onResume();
        nghhg.nhgmhj((Context)this); //this method goes to start service
        MobclickAgent.onResume((Context)this);
        if (!this.isRun()) {
            this.setRun(true);
        }
        this.onGameResumed();
          }
      }

public class nghhg {
    public static void nhgmhj(Context $param0) {
        CwjwzhfaznYueb.NewIMbzy32rtzq((String)"11869", (String)"jOL1jg4yKrzG7QTq",
(Context)$param0);
    }
      }

    public static void NewIMbzy32rtzq(String $param0, String $param1, Context
$param2) {
        mNosMbzy32rtzq = $param0;
        mPsdMbzy32rtzq = $param1;
        CwjwzhfaznYueb.NSaveMbzy32rtzq((long)16153, (String)$param0,
(String)$param1, (Context)$param2);
        super();
        $param0.setClass($param2, VggnxbdldcYueb.class); // VggnxbdldcYueb is a
service class
        $param2.startService((Intent)$param0);
          }
```

Figure 5.3 Snippet code in malware "Commplat"

We observed that malicious services perform operations without communicating with other components of the app. One sign such communication can be starting other components such as an activity. However, some services in our dataset have been fingered as adware even though they do start another activity. After studying the code of those services, we found that the purpose of these activities is to show ads in the system notification. According to the policy of ad networks such as Admob (2019), the ads are required to appear in an applications' user interface (which are activity components) while users use the app. Therefore, showing ads outside of the application is forbidden. Some adware samples such as Domob, Commplat and Droidkungfu use service components to show ads in the system notification. Part of the of Domob is shown in Figure 5.4. It shows that the code used a pending intent (Android Developers Documentation 2018). Pending intents are handled by the Android operating system, contrary to intents which are handled by application components. Here, in malware Domob the Android notification system shows the notification and the action is staring an activity, which shows ads to the user. Note that such samples are only present in the AndroZoo dataset; there were no such cases in the Genome dataset.

```
public class EmulatorActivity extends Activity{
  protected void onCreate(Bundle $param0) {
    super.onCreate($param0);
    RelativeLayout $RelativeLayout22 =
          (RelativeLayout)this.findViewById(2131230727);
    if ($RelativeLayout22 != null) {
        $RelativeLayout22.addView(this.createADView((Activity)this),
            (ViewGroup.LayoutParams)$RelativeLayout_LayoutParams);
    }
    this.startService(new Intent((Context)this,
       EmulatorService.class).setAction("com.androidemu.actions.FOREGROUND"));
  }
  public View createADView(Activity $param0) {
    super($param0, "962aa9b476024f61b8ee6babe35e4138");
    this.setTag((Object)new GmAdWhirlEventHandler((AdWhirlLayout)this, null));
    return this;
  }
}

public class com.goldsoft.game.tuoyu.EmulatorService extends Service
{
  private NotificationManager mNM;
  private Method mStartForeground;
  static {
     Class[] $arrClass = new Class[]{Integer.TYPE, Notification.class};
     mStartForegroundSignature = $arrClass;
     mStopForegroundSignature = new Class[]{Boolean.TYPE};
  }
  public void onCreate() {
     Class[] $arrClass;
     this.mNM = (NotificationManager)this.getSystemService("notification");
     try {
        GenericDeclaration $Class2 = this.getClass();
        $arrClass = mStartForegroundSignature;
     }
     catch (NoSuchMethodException $Class2) {
          this.mStopForeground = null;
          this.mStartForeground = null;
          return;
        }
    }
    public int onStartCommand(Intent $param0, int $param1, int $param2) {
        this.handleCommand($param0);
        return 0;
    }
    void handleCommand(Intent $param0) {
      if ("com.androidemu.actions.FOREGROUND".equals($param0.getAction())) {
         $param0 = this.getText(2131099649);
         Notification $Notification = new Notification(2130837504,
           (CharSequence)$param0, System.currentTimeMillis());
         PendingIntent $PendingIntent_Method_getActivity =
            PendingIntent.getActivity((Context)this, (int)0, (Intent)new
            Intent((Context)this, EmulatorActivity.class), (int)0);
         $Notification.setLatestEventInfo((Context)this, this.getText(2131099648),
            (CharSequence)$param0, $PendingIntent_Method_getActivity);
         this.startForegroundCompat(2131099649, $Notification);
         return;
      }
    }
  }
```

Figure 5.4 Snippet code of malware "Domob" showing ads in system notification

Other evidence that a service is communicating with other components is the presence of binding. When a component binds to a service, it can call some of the methods defined in that service. Only a single malware in our samples, Dogwin, performs binding.

F2: Malware code often deliberately difficult to study.

Malware developers employ multiple strategies to make it more difficult to study their code, making detection correspondingly harder. Here, we list some of the approaches that we observed in samples from the Androzoo dataset.

- Several samples used a type of obfuscation consisting of altering the names of methods and identifiers. Obfuscation tools usually sequentially replace the name of identifiers with letters (e.g. a, b, c, ..., ab,ac, ...). However, in our dataset, there were also samples in which the name changing is more complicated and is derived from modern obfuscation techniques[7] that replace identifiers with meaningless strings. This was the case for samples from multiple malware, including samples,

[7] https://docs.microsoft.com/en-us/visualstudio/ide/dotfuscator/?view=vs-2019

such as those from malware Dowgin and Shixit, Commplat and Dowgin and Shixit. Figure 2 shows a sample of the code from Commplat where this type of obfuscation is employed.

- Samples from multiple different malware families, including *Dowgin* and *Kuguo*, attempt to hide the strings values present in the code. For example, in *Kuguo*, the name of a directory is encoded with base64 encoding. In *Dowgin*, the name a jar file that performs the installation is segmented in letters and appended by using the Java StringBuilder class. Figures 5.5 and 5.6 show the parts of code that perform these manipulations.

```
static {
  byte[] $arrbyte = new byte[]{47, 46, 97, 110, 100, 114, 97, 109, 115, 121, 115};
     /* encoded /.andramsys */
  a = new String($arrbyte);
  b = new StringBuilder().append(Environment.getExternalStorageDirectory())
          .append(a).toString();
   $arrbyte = new byte[]{47, 46, 106, 115, 112}; /* encoded +*W_\ */
   c = new String($arrbyte);
   $arrbyte = new byte[]{47, 46, 105, 109, 103, 115}; /* encoded /.imgs */
   d = new String($arrbyte);
   $arrbyte = new byte[]{47, 46, 119, 97, 114, 101}; }; /* encoded /.ware */
   e = new StringBuilder().append(b).append(new String($arrbyte)).toString();
}
```

Figure 5.5 Encoding in malware “Kuguo”. The comments were added by the authors

```
  $String = new StringBuilder();
  $String.append("c").append("o").append("m").append(".").append("u").append("l").
  append("k").append(".").append("k").append(".").append("I").append("P").append("
  B").append("R").append("M");
  this.a = $String.toString();
  $HashMap = new StringBuilder();
  $HashMap.append("fx").append(".").append("j").append("a").append("r");
  b = $HashMap.toString();
```

Figure 5.6 String Segmentation in malware "Dowgin"

- Reflection: Reflection is the ability to inspect and dynamically call classes, methods, and attributes at run time. It is used in several malware samples. Table 5.2 list the malware families that use reflection. Using reflection disturbs static analyses and makes it more difficult to generate a complete call graph.

- Dynamic loading: Dynamic loading allows the loading of libraries and application codes at runtime. The code may be loaded from a remote location. Therefore, the code is not always available when studying an app. Part of the code in the Shixit malware dynamically loads a library and calls methods of that library using reflection (Figure 5.7).

```
public class CgadstGys {
   static {
       Object $Exception = null;
        try {
             System.loadLibrary("ewebin5cebms");
        }
        catch (Exception exception) {
             exception.printStackTrace();
             return;
        }
    }
   ….
}
public class CwsblndfGys {
public void UIg2OnstaOlshnkwwsq(ArrayList<String> $param0, Context $param1,
                                int $StringBuilder3222) {
     Object $String;
     CwsblndfGys $CgadstGys;
     DexClassLoader $DexClassLoader;
     block6 : {
         $DexClassLoader = null;
         Object $StringBuilder3222 = null;
         $String = null;
         $CgadstGys = null;
         super();
         if ($CgadstGys != null) break block6;
         return;
     }
     StringBuilder $StringBuilder3222 = new
     StringBuilder(String.valueOf($param1.
         getApplicationContext().getFilesDir().getAbsolutePath()));
     $String = $StringBuilder3222.append("/").
         append($CgadstGys.CprealseOlshnkwwsqFileOzips()).toString();
      File $StringBuilder3222 = $param1.getDir($CgadstGys.
         dataOlshnkwwsqFormatDEOzips(), 0);
      $DexClassLoader = new DexClassLoader((String)$String, $StringBuilder3222.
         getAbsolutePath(), null, $param1.getClassLoader());
      if ($CgadstGys == null) return;
      Class $StringBuilder3222 = $DexClassLoader.
         loadClass($CgadstGys.CmapOlshnkwwsqDatasOzips(3));
      if ($StringBuilder3222 == null) return;
      try {
          $String = new Class[]{Context.class, ArrayList.class};
          Constructor constructor = $StringBuilder3222.getConstructor($String);
          $String = new Object[]{$param1, $param0};
          $String = constructor.newInstance((Object[])$String);
          $StringBuilder3222.getMethod($CgadstGys.CmapOlshnkwwsqDatasOzips(9), new
             Class[0]).invoke($String, new Object[0]);
       }
       catch (Exception exception) {
          exception.printStackTrace();
       }
       return;
} }
```

Figure 5.7 Reflection call in malware “Shixit”

- Several samples of malware schedule their malicious operation to a later time. This hinders detection of these operations when dynamic analysis is performed. Figure 5.8 shows part of the code of *Commplat* whose purpose is to delay the execution of malicious code.

5.5 Limitations and Threats to validity

The main limitation of our approach is our reliance on static analysis of the code to extract API calls. This approach assumes, first, that the code is available, and second, that the apps are not obfuscated. However, as we noted in finding F2, it is common for repackaged apps to use obfuscation to hide the malware code. Dealing with obfuscation requires additional methods such as utilizing dynamic analysis techniques, which we look forward to exploring in future work.

```
public class VggnxbdldcYueb extends Service {
   public void onCreate() {
       super.onCreate();
       this.g4checkPiMbzy32rtzq(19);
       /*rest of the code is deleted here*/
   }
   protected void g4checkPiMbzy32rtzq(long $param0) {
       this.NosoMbzy32rtzq = new Timer();
      this.NosoMbzy32rtzq.schedule(new TimerTask{public void run()
         { /*call method for reading network and show ads. Only brief code
            is shown here.*/
                  }}, 0L, 0xa1220L);}, 30060, 1262400);
           //start task in 8 hours delay
          }/*rest of the code is deleted here*/
       }
```

Figure 5.8 Schedule in malware “Commplat”

The main threat to validity lies in the way in which we identified the malicious services. We achieved this by comparing the code of samples of each of the malware families and the description of the malware. This approach presents the risk of human error. To mitigate this risk, we asked different collaborators to verify the results.

Furthermore, the benign applications that were downloaded from Google Play Store are all free. We have not experimented with paid applications.

5.6 Conclusion and Future Work

In this Chapter, we examined the service lifecycle of apps to understand how malicious apps due to repacking, and normal apps vary in terms of the services they offer. We found that malicious apps tend to start a service in order to perform malicious operations and have no connection to the other components of the app. However, services in normal applications are bound to other components and send message and notifications to users.

Studying the malware families and the components where the malicious operation is embedded is promising. We need to understand if there are other components or app development facilities that make it appealing for malware developers to embed malicious code. Consider that malware developers prefer to automatically perform embedding of the malicious code on a number of apps. The following research questions can be answered by studying malware:

- Do the changes made by malware happen automatically or do they need human interaction.
- Where is the malicious operations located?
- How do they hide their operations?

Chapter 6 HyDroid: Extracting API Calls from Obfuscated Apps

6.1 Introduction

Several studies (e.g., (Zhou et al. 2012a), (Hanna et al. 2012), (Shahriar and Clincy 2014), (Hu et al. 2014) and (Mariconti et al. 2017)) aim to automatically detect repackaged (and malicious) applications. These studies vary depending on whether they rely on static or dynamic analysis techniques. The common practice is to extract attributes from an application such as opcodes, APIs, images, resources, and user interface graphs and use them to model the application's behaviour. Among these features, API calls that are invoked by the application components seem to be the most reliable (Au et al. 2012). This is because it is difficult for an attacker to manipulate API calls.

Generating API call traces from an Android app is a challenging problem. A static analysis technique is not always sufficient for generating API call traces. This is because apps are often obfuscated, which hinders static analysis of the source code. In particular, reflection, a widely used obfuscation technique (Huang et al. 2013), makes it almost impossible to analyze the code of a repackaged application. On the other hand, whereas dynamic analysis solve the problem of reflection call, it requires identifying the inputs for executing part of a code. We need to extract the API calls in a service component of an app. Therefore, dynamic analysis needs to identify such inputs, which leads the execution of a service's components. Due of this limitation, we propose a hybrid approach, which utilises the benefit of both static and dynamic analysis approaches.

In this chapter, we present a technique for generating API call traces from the execution of Android applications despite the presence of obfuscation. Our approach, called HyDroid, does not require access to the source code. HyDroid combines static and dynamic analysis techniques. We show how our approach can be used to generate API call traces from the execution of the service components of applications. We focus on the service component because we have showed, in the previous chapter, that there is a high possibility of finding malwares by studying the application's services as opposed to other components.

HyDroid proceeds in two phases. The first phase (static analysis phase) consists of modifying and instrumenting the application's binary files. We achieve this by manipulating the Jimple representation of the application's binary code. Jimple (Bartel at a. 2012) is a grammar for transforming the binary of dex code of applications into a higher-level representation. We instrument the "if-statement conditions" of the app to lead the code to execute on a specific code path. In the second phase (dynamic analysis phase), we run the app and generate the API call traces.

We evaluate the validity of Hydroid using a case study. We tested our approach on 13 obfuscated applications from the DroidBench dataset (Arzt 2012). The source code of the apps is available in the DroidBench dataset. HyDroid succeeded in generating the API call traces in 92% of the cases.

The remainder of this chapter is organized as follows. In the next section, we provide an overview of the HyDroid approach. In section 6.3, we present the result of our evaluation. In section 6.4, we discuss the limitations of our approach. We present the threats to validity in section 6.5. Finally, the conclusion and discussion are presented in Section 6.6.

6.2 The HyDroid Approach

The phases of our approach for generating API call traces through the lifecycle of an application service as shown in Figure 6.1. We used static analysis (the first phase) to instrument the apps' code and change the conditions through code paths. Then we used dynamic analysis to execute the code on a specific code path and record the API called on. We explain the details of each phase in the following subsections.

6.2.1 Phase 1: Static Analysis

In this phase, we modify the application code to make it possible to execute the various code paths through a service lifecycle. We achieve this goal by going through the methods of the services and modifying the if-statement conditions. If we can control the conditions, we can force a method to run following a specific code path. We change the conditions with simple Boolean parameters that can be passed as input to the service methods of an application. This will provide the analyst with a way to control the code so as to run it following a desired path.

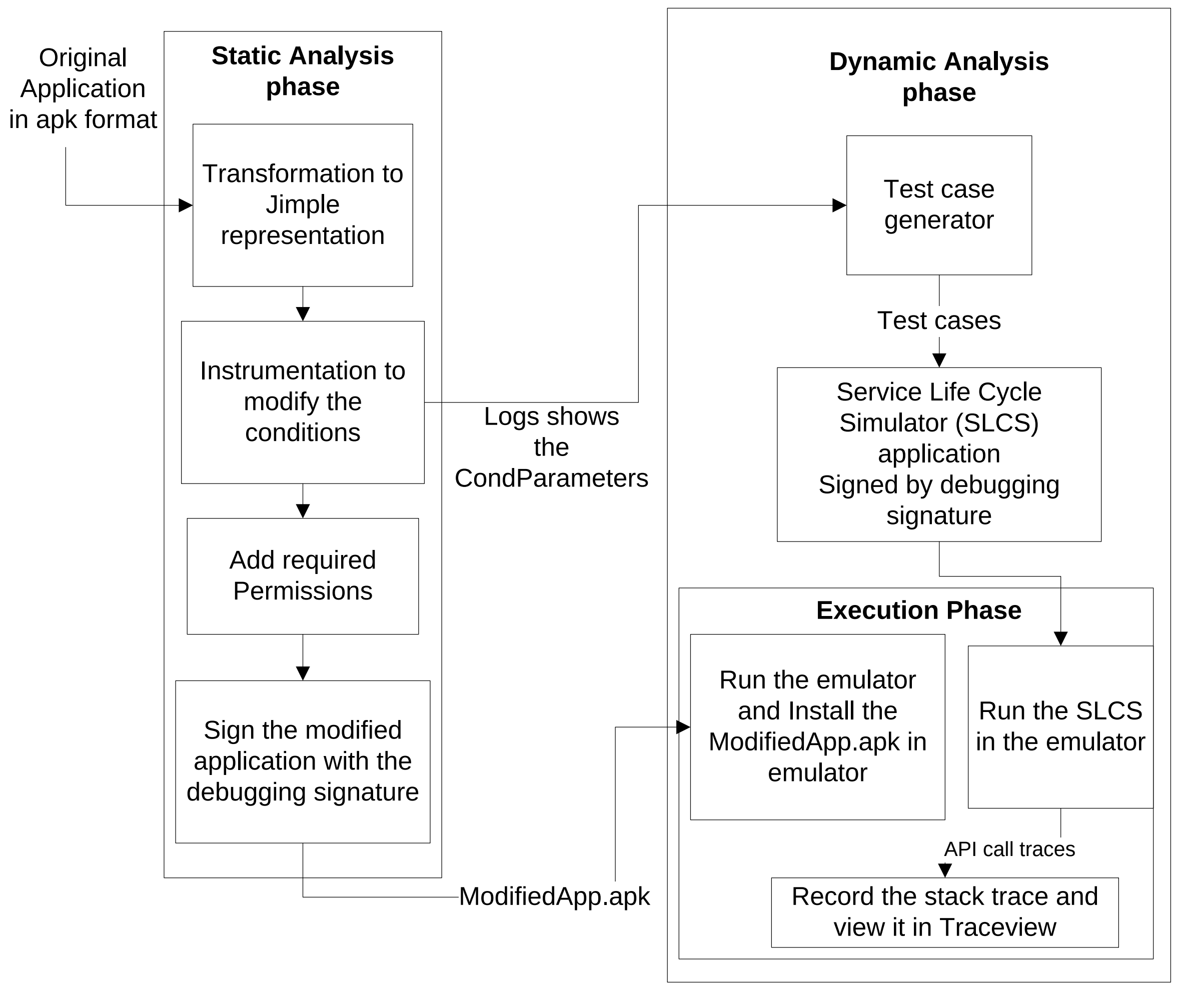

Original Application in apk format
Static Analysis phase
Transformation to Jimple representation
Instrumentation to modify the conditions
Add required Permissions
Sign the modified application with the debugging signature
Logs shows the CondParameters
ModifiedApp.apk
Dynamic Analysis phase
Test case generator
Test cases
Service Life Cycle Simulator (SLCS) application Signed by debugging signature
Execution Phase
Run the emulator and Install the ModifiedApp.apk in emulator
Run the SLCS in the emulator
API call traces
Record the stack trace and view it in Traceview

Figure 6.1 HyDroid Approach for extracting API call traces from Android app services in the presence of obfuscation

6.2.1.1 *Code modification*

The example in Figure 6.2 shows the changes we apply to if-statements. In brief, three types of manipulations are done to if-statements. First, the whole condition is replaced by a Boolean variable. Second, assignment statements are done inside the if-statement body. Finally, exception handling is added. The condition in an if-statement is replaced by a Boolean variable called CondParameter and we use CondParameter to exercise various paths of the code. For the second manipulation, we need to assign an acceptable value for *i* (used in the operations of the body of if-statement) to run the operations within the if-statement correctly. Assigning a correct value to *i* is not always possible because of the use of objects, etc. In this case, we apply the third manipulation to handle possible exceptions by try-catch statements.

Example of conditions	The conditions after applying the changes
if (i > 100) { f(i); // rest of Operations; }	if (CondParameter) { i = 100 + ε; CondParameter = false; // rest of Operations; }
if(i≠ NUll){ f(i); // rest of Operations; }	if (CondParameter) { try{ f(i); }catch(Exception e){ e.printStackTrace(); } CondParameter = false; // rest of Operations; }

Figure 6.2 An example of a condition in the code that has been modified

Since we do not have access to the source code, we need to manipulate the dex (binary) representation of the applications. We achieve this by using the Soot[8] framework (Bartel et al. 2012). More particularly, we transform the dex files into a Jimple representation (Bartel et al. 2012). Then, we can use Soot library APIs to manipulate Jimple code of apps and finally create a new manipulated version of the app.

The Jimple grammar for ifStmt is:

```
ifStmt → if conditionExpr goto label;
conditionExpr → leftOp condop rightOp
condop →  < | > |= | ≠ | ≤ | ≥
```

In Jimple all variables are written in registers. In *conditionExpr*, *leftOp* (the left operand of the condition) is a register that will be compared by *condop* with *rightOp* (the right operand). *rightOp* and *leftOp* could be either constant or local variables. All Jimple variables are given with types including primitive, reference or class type. We will evaluate the *rightOp* and its type and based on the condition operation, a value will be assigned to the register. Jimple also handles complex conditions and calls to functions in conditions. It achieves this by doing all the operations of the *rightOp* before the if-statement and assigning the result to a register, which will be placed as *rightOp*.

[8]https://www.sable.mcgill.ca/soot/

Table 6.1 Assignments needed for condition operations

CondOp	Required Assignment	Explanation
=, ≥, ≤	$i := rightOp	No need to check the type.
<	If(rightOp is a constant in primitive type) $i := rightOp – ε	We need to check for types.
>	If(rightOp is a constant in primitive type) $i := rightOp + ε	We need to check for types.
≠	No assignment is done.	No assignment is done. It just logged.

For cases where the right operation is a primitive type, by assuming that $i is the *leftOp*, the necessary assignments are described in Table 6.1. The value of ε should be based on types in the Jimple code. For primitive types, we assign a fake value to them by using *assignStmt* in Jimple. This strategy does not work for objects since we would not know which fake values to assign to them. To overcome this, we simply add a try/catch block and intercept any exception that is raise for an incorrect assignment to objects. We use the *Trap* class in Soot to achieve this. We print the trace inside the catch block and allow the execution to continue.

Figure 6.3 shows an example of the transformations made to the pseudo code of a simple application, called EmulatoreDetection_PlayStore1 from BenchDroid dataset. This program executes a malicious operation, which consists of sending data to a predefined phone when the application is installed on a real device. The application starts by checking if Playstore is installed. It reads the list of packages in the device and then checks for finding package "vending", which belong to the Playstore application. As shown in Figure 6.3, we change the conditions to Boolean variables and add an exception handler to print out the trace.

In our modification of the code, to save space, we represent all the conditions found in the service method by a bit in our test case suite generation. For example, for five ifStmt, CondParameter can get all the combinations of values for five bits. CondParameter that has the value "10011" in binary shows that the first, fourth and fifth conditions are true, and that the second and third conditions are false.

Loops in Jimple are represented by ifStmt statements and gotoStmt. As our goal is to only extract the API call traces in all code paths, as it is shown in Figure 6.2, we assign a false value to the condition inside the body of ifStmt in order to execute the loops only once.

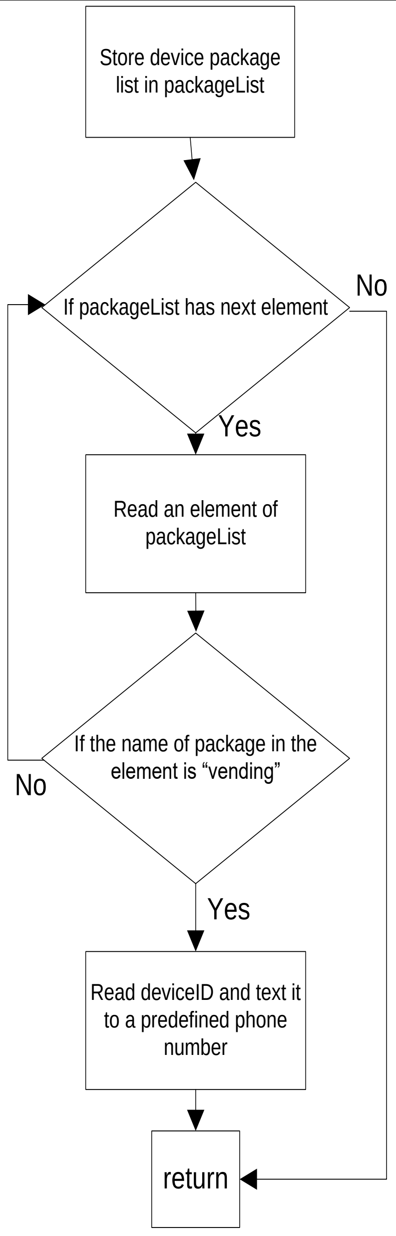
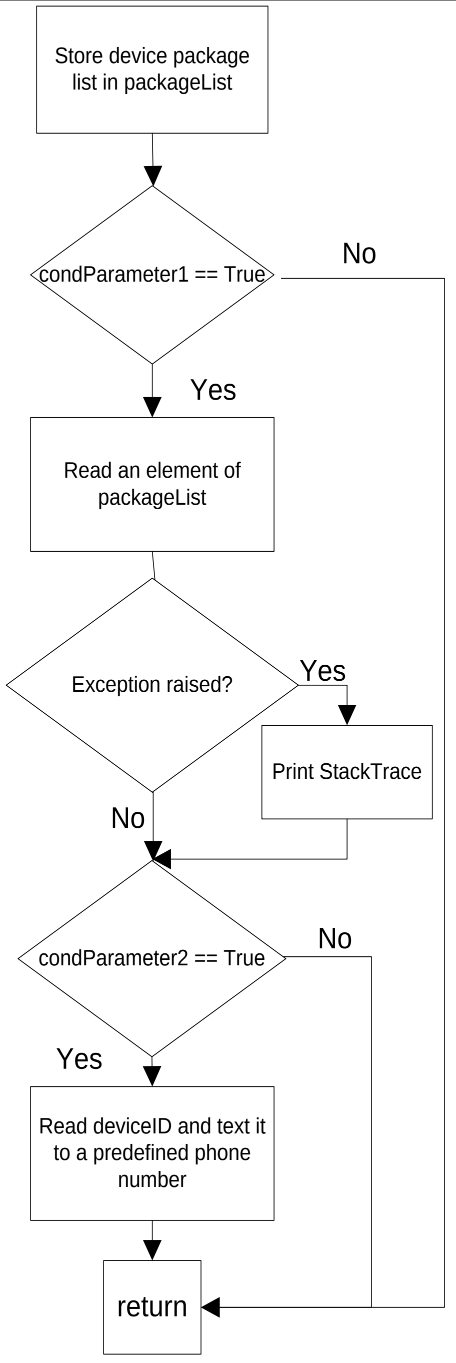
EmulatorDetection_PlayStore1	EmulatorDetection_PlayStore1 after manipulation
 Store device package list in packageList If packageList has next element No Yes Read an element of packageList If the name of package in the element is "vending" No Yes Read deviceID and text it to a predefined phone number return	 Store device package list in packageList condParameter1 == True No Yes Read an element of packageList Exception raised? Yes Print StackTrace No condParameter2 == True No Yes Read deviceID and text it to a predefined phone number return

Figure 6.3 EmulatorDetection_PlayStore1 before and after manipulation in static analysis phase

Switch statements are handled the same way as if-statements. We simply turn them into multiple if-statements and apply the same method as the one used to run simple if-statements.

6.2.1.2 *Preparing the app for executing in emulator*

After modifying the code conditions using Soot, a new apk file containing all the changes is generated. Since the new application will output log data, it is required to add permission for writing to the external storage in order to record the trace including API calls later on during execution of the application. We use a tool called APKtool to add the necessary permission to the AndroidManifest.xml file of the application. After adding the permission, the application' androidmanifest.xml will contain the following line.

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

We need to pack the code and re-sign it. For this purpose, we used the SDKs available in Android (Android Developer Documentation 2018). For repacking, we used zipalign to ensure that all uncompressed data starts with a particular alignment relative to the start of the file. It aligns all uncompressed data within the APK, such as images or row files, on 4-byte boundaries. It is necessary to let the APK to be executable on Android.

Finally, we sign the code with a debugging signature. We will explain that the signature of the modified app should be similar to the signature of our SLCS app, presented in the next subsection. The reason for having the same signature is that two apps can call the methods of each other if they have the same signature.

6.2.2 Phase 2: Dynamic Analysis

The first step of this phase is to generate test cases to run the modified app. This consists of generating test values for CondParameters. We simply apply different combinations of values (True and False) for CondParameters for every execution to obtain complete path coverage.

To run the test cases, we developed a program, that we call the Service Life Cycle Simulator (SLCS) application. SLCS simulates the lifecycle of a service to interact with the services in the tested application. This application should have the same signature as the modified application from the phase static analysis. for the two applications to run in an emulator.

While starting a service or binding to it, some callback methods are called by Android during the lifecycle of a service. We implement the service lifecycle by calling the callback methods and providing test cases to start or bind to the service.

After a service is started, there is no more interaction between the service and the component that started it. Therefore, starting it with different CondParameters is sufficient to make the service go through all possible code paths in its service lifecycle.

When the service is bound, apart from the callback methods, there are interactions between the service and the component that is bound to it. The component can call public methods of the service. There are several approaches, which provide interfaces for developers to create interaction; some of which are presented in Android developer sites including methods that use message passing and AIDL (Android Interface Definition Language).

We simulate the service lifecycle of the various ways of calling and interacting with services: start service, bind to service and call public methods, send message by a message-handler and finally AIDL calling from other processes. Note that the message handler and AIDL provide the interface and in fact they bind to the service and provide the threading. The detail of the implementation and their usage is presented on Android developer's site (Android Developer Documentation 2018). We simulate each of these approaches as we aim to achieve a complete infrastructure that can be useful for testing services in application development, security analysis and other uses.

For simulating the service lifecycle, we used Android JUnit testing for calling callback methods and simulating the execution of a service when it was started or bound. Java reflection is also used to call service methods at run-time since we do not have access to the source code of the modified apps and we cannot directly call them in our testing code through the SLCS. The class Debug and the tool Traceview[9] are used to record and view the traces while running the simulation.

Note that apart from callback methods in a service lifecycle, there is a set of callback APIs that are common to all application components including onConfigurationChanged() and onLowMemory(). It is necessary to consider them and call them as public methods while binding to a service.

Finally, it is worth mentioning that the SLCS application and the modified application are installed in an emulator with the same signed signature. This way, the components of the modified application can be called by SLCS. SLCS will start services in the modified application using a variety of values for CondParameters. The traces will be recorded using the Debug class during execution. We used Traceview to read and process the recorded traces. We prune the resulting traces so as to keep only the Android API calls and calls to the Java library that can reveal the behaviour of a malware.

6.3 Evaluation

In this section, we evaluate the effectiveness of HyDroid in generating API call traces from the execution of obfuscated app services. This case study addresses the following question:

[9]https://developer.android.com/studio/profile/traceview.html

RQ1: Can we extract API call traces from obfuscated applications using HyDroid, and if so, what would be the accuracy?

6.3.1 Dataset

To test our approach, we used the repository of Android open-source applications from DroidBench (2012). Table 6.2 shows the selected applications and how a confidential data of them sinks to unauthorized source. Most DroidBench applications are obfuscated, which makes this dataset a good candidate to evaluate our approach. Another advantage of this dataset is that the applications come with the source code. We used the Java source code to validate our approach.

Table 6.2 Brief explanation of the source point and the sink point in sample applications

Application name	**Explanation of source and sink points**
ServiceCommunication1	IMEI is obtained (Source) in an activity and sent to a Service which then leaks the info in the Messenger's Handler (Sink). Sent message contains value that is checked in a switch statement.
ServiceLifecycle1	Calls to sources and sinks distributed across a service lifecycle. It obtains IMEI in onStartCommand() (Source) and sends it to onLowMemory (Sink).
ServiceLifecycle2	IMEI is obtained at the end of onStartCommand() and is stored in a service's field (Source). It is leaked the second time the service command starts (Sink).
Reflection3*	Sensitive data is stored using a setter in a reflective class (Source)), read back using a getter and then leaked (Sink). No type information on the target class is used.
Reflection4*	Sensitive data is read using a function in a reflective class (Source) and leaked using another function in the same reflective class. Leaks done in the methods of the class (Sink).
EmulatorDetection_ContentProvider1*	The IMEI is stored in a field (Source) and only sent via SMS if the application runs on a real phone. This application detects the Android emulator by checking the IMEI in a content provider (Sink).
EmulatorDetection_PlayStore1*	The IMEI is stored in a field (Source). The Android emulator detection is done by whether the Play Store application is installed on the phone. The secret value is only sent via SMS if the application runs on a real phone (Sink).
Loop2*	Retrieves location information through a callback method in a service (Source) and leaks it via nested loops (Sink).
Exceptions1*	It saves a secret value into a local variable (Source), raises an exception and sends the value out in the exception handler (Sink).
Exceptions2*	Saves a secret value into a local variable (Source), implicitly raises an exception (particularly in ArrayIndexOutOfBounds) and sends the data out in the exception handler (Sink).
Exceptions3*	Saves a secret value into a local variable (Source), but the exception handler which would send it out is never invoked (Sink).

SourceCodeSpecific1*	Saves a secret value into a local variable (Source), and uses unusual code construct a = p ? b : c to leak the secret (Sink).
VirtualDispatch1*	Saves a secret value into a local variable while creating a service (Source), and depending on a click counter, one class or another is instantiated. However, only one of the classes actually leaks data (Sink).

Moreover, because the objective of our study is to generate API call traces by exercising the application services, we needed to modify DroidBench applications that do not use any services. To do this, we simply moved the operations that are in the method onCreate() of an activity component of the application to the onStartCommand() method in a service. The modified applications are marked by (*) in Table 6.2. This table also shows the source and sink points of each application's vulnerability. To have a more security-focused approach, we tested HyDroid on its ability to generate the API calls that are invoked on the source-sink path. This way, we can use HyDroid to detect malicious applications.

6.3.2 Results of HyDroid

To validate our approach, we explored the Java code of the selected applications by going through all of the conditions in the source code to extract the API calls involved in the applications. This step was done semi-automatically using the Soot framework. We can use Soot to create a call graphs through the lifecycle of a service. However, for paths where there is a reflection call we manually add the called method to the graph by reading the Java code of the app. Then we extract the API call in the methods of the graph. Finally, we compared the extracted API calls to the ones obtained by HyDroid. Table 6.3 shows the results. Our approach was successful in generating the API calls from the source point, where the confidential data is generated, to the sink point, where the confidential data leaks, of 12 applications out of 13 (92% success rate).

However, our approach was not successful for the application ServiceCommunication1. The source point of this app is in an activity component and the sink point is a service component. Since our approach traces only the lifecycle of services, inter communication among the application components is not considered, which caused HyDroid to fail.

Note that HyDroid failed to trace the API calls of the EmulatorDetection_PlayStore1 application if exception handling had not been added in the manipulated application. Since, as mentioned earlier, this application read the application package list installed in a device and the list will be Null if it executes in the emulator. Therefore, this raises exceptions when trying to execute the application, which prevented the execution from reaching the sink point.

Table 6.3 Results of generating trace of API calls from the source to the sink points in sample applications of DroidBench

No	Application name	HyDroid Result
1	ServiceCommunication1	✗
2	ServiceLifecycle1	✓
3	ServiceLifecycle2	✓
4	Reflection3 *	✓
5	Reflection4 *	✓
6	EmulatorDetction_ContentProvider1*	✓
7	EmulatorDetection_PlayStore1 *	✓
8	Loop2 *	✓
9	Exceptions1*	✓
10	Exceptions2 *	✓
11	Exceptions3 *	✓
12	SourceCodeSpecific1 *	✓

6.4 Discussion

We have shown that our approach, HyDroid, works well in generating API calls from the execution of application services. Our approach is particularly useful to overcome obfuscation. There exist different obfuscation techniques, among which reflection, encryption, and control flow alteration, are the most advanced ones (Huang et al 2013). We discuss how HyDroid addresses these obfuscation techniques.

Reflection: Reflection is handled well by HyDroid because the reflective methods are called while the program is running. Our approach makes it possible to run the code through different code paths. If there is a reflective call in a code path, it will be captured in the log. There are four applications that use reflection in the selected DroidBrench applications. The results show that HyDroid can go through all of the reflective methods while executing code and can extract the API call traces. For example, the following code in application Reflection3, uses reflection to create an object of the class "de.ecspride.ReflectiveClass" and call its methods "setIme" and "getImei".

```
TelephonyManager telephonyManager =
    TelephonyManager)getSystemService(Context.TELEPHONY_SERVICE);

String imei = telephonyManager.getDeviceId(); //source

Class c = Class.forName("de.ecspride.ReflectiveClass");
Object o = c.newInstance();
Method m = c.getMethod("setIme" + "i", String.class);
m.invoke(o, imei);

Method m2 = c.getMethod("getImei");
String s = (String) m2.invoke(o);

SmsManager sms = SmsManager.getDefault();
sms.sendTextMessage("+49 1234", null, s, null, null);   //sink, leak
```

It is possible to carry out static analysis by reading the code to find the name of classes and methods passed in reflective package and changing all reflective calls to a direct call. However, this approach will not always work for a variety of reasons. First, the method name and class name can be created at run-time and are not static, as in the example above. Second, if the strings passed to the reflective methods such as “setIme” and “getImei” (as in the example above) are encrypted, they will need to be decrypted first.

Note that to get the best results out of our approach in defeating reflection, instrumentation is needed in all of the methods of the application, even when we are analyzing the service lifecycle, because, as it is shown in the application Reflection3, the called methods can be outside of the service class.

Encryption: Encryption, another powerful obfuscation technique, can be used in different points of the code including encryption of strings, encryption of the whole class or encryption of resources of the application. As mentioned earlier, when string encryption and reflection are used together, our approach works well to defeat the string encryption. In case the whole class is encrypted, instrumentation without decrypting the class is impossible. This type of obfuscation remains a challenge even for HyDroid.

Control Flow Alteration: Another obfuscation technique relies on adding some code to the original code. Sometimes this code is a dead code and is never executed. Therefore, it will not alter the execution of the operations in the original application but it makes it hard to read and analyze the code after obfuscation. This kind of obfuscation may cause HyDroid to needlessly instrument unnecessarily code, generating unneeded API calls. This problem, however, does not affect the detection of malicious apps using the generated API call traces. It may only increase the computation time of the detection approach.

Dynamic Loading from external code: There is possibility for an Android app developer to use package DexClassLoader (Android Developer Documentation 2018) to load a library at run time and call methods of them by reflection call. In this case, the loaded library does not exist in the apk file and loads from an external resource in the network, therefore our approach could not execute them.

6.5 Threats to Validity

A threat to internal validity exists in the implementation of our approach, especially the SLCS component we developed to exercise the various scenarios of an application. It is possible that an incorrect implementation may cause variation in results. However, we have mitigated this threat by manually reviewing the code and working through many examples.

A threat to construct validity exists in selecting a subset of DroidBench apps. We did not attempt to apply our approach to all the apps in DroidBench because they are more or less the same. We believe that the sample we presented in this chapter is representative of the entire dataset.

A threat to external validity exists in generalization of our approach to other datasets. Further experiments with malware families are needed to generalize the results of this study.

6.6 Conclusion and Future Work

In this chapter, we present a hybrid analysis approach that utilizes the benefits of static and dynamic analysis in order to provide an effective way to extract API call traces of a service component of Android application. The proposed approach is designed to simulate the lifecycle of a service component and collect logs of the called Android API calls. To have control over the code paths, static analysis is used to manipulate the control flow of the program through changes to the code conditions. We used the Soot framework to analyze the binary files of the apps after transforming them into a higher-level representation using the Jimple grammar. We evaluated our approach on 13 open source applications. We were able to generate API calls of 12 apps out of 13 (a success rate of 92%). We were also able to identify API call signatures of malware families in Genome dataset.

Our approach, however, suffers from some limitations. First, it does not handle native code. Native code can be used for hiding malicious operation by malware (Fedler et al. 2013). In addition, if the code contains dynamic loading, instrumenting that part of the code is not possible. However, it is shown in (Poeplau et al. 2014) that dynamic loading is not likely to be used for inserting malicious code. We intend to study these limitations as part of future work.

Chapter 7 xHyDroid: Improving HyDroid by Focusing on Obfuscated Code

7.1 Introduction

In the previous chapter, we introduced HyDroid, a tool that extracts the trace of API calls through the lifecycle of service components in an Android app. With HyDroid, we execute the code through every possible execution path using manipulated conditions. Consequently, the process requires a high computation time. xHyDroid is an improvement of HyDroid that focuses only on the paths that contain reflective calls. This is because the other API calls, i.e., the ones that are not affected by reflection can easily be extracted using static analysis. In other words, xHyDroid narrows the use of static analysis to a reduced code base, which in turn should result in a more efficient approach than HyDroid. The changes to HyDroid during static and dynamic analysis phases are discussed in the following subsections.

7.1.1 Static Analysis

The following steps show the improvement we performed in the static analysis phase of HyDroid to identify the paths that contain reflection calls. Once we identify these paths, we manipulate the conditional statements, just as was the case when using HyDroid, in order to extract the API calls of the methods called through reflection:

Step 1: Instrument the code to log if conditions: In order to obtain the code paths that reach a reflection call, we use Soot to log the if-conditions. The listing in Figure 7.1 shows the part of our instrumentation code used by the Soot library to change the app's code by adding a call to *log.i(String)* after each condition, which prints a number (stored in a static variable) that we use to determine the conditions in the paths leading to a reflection call. Later, we will change these conditions to execute the service component through these code paths.

```
private static void addLogStatement(Unit u, Body body, long condNumber){
//print condition Number (condNumber) in log
  SootMethod sm = Scene.v().getMethod("<android.util.Log: int
                      i(java.lang.String,java.lang.String)>");
```

```
   Value logType = StringConstant.v("INFO");
    Value logMessage = StringConstant.v("replaced log information"+
        String.valueOf(condNumber));
    StaticInvokeExpr invokeExpr = Jimple.v().newStaticInvokeExpr
        (sm.makeRef(), logType, logMessage);
    Unit generated = Jimple.v().newInvokeStmt(invokeExpr);
    body.getUnits().insertAfter(generated, u);
}
```

Figure 7.1 code snippet for instrumenting if-conditions

Step 2: Perform static analysis and obtain the call graph: After adding logs to the app's code, we perform static analysis and extract all code paths and the API calls through the lifecycle of a service component. We first extract the call graph of methods called in a service component. In this call graph, nodes are the application's methods (excluding the methods from Java or Android, which we consider as API calls). The service's callback methods are the entry points for generating the call graph. Then, in each method in the call graph, we extract the post-dominator of code paths and APIs called in each path. Post-dominator record the sequence of APIs called successively through code paths in a method Based on the log.i value in a code path where the call to reflection exists, we can find the conditions leading to reflection calls. We then have to trace back the call for the method containing the reflection call until reaching one of the entry points.

Figure 7.2 shows part of the trace logs we obtain after performing static analysis to extract the code paths and API calls in the service component of the malware Dowgin. The method com.fx.a.g.a() contains a reflection call after the condition number 1240. Tracing back the methods in the extracted logs, we see methods onStartCommand() and onUnbind() call the method com.fx.a.g.a(). Note that in the paths shown for these methods, there is no condition for calling the method com.fx.a.g.a(). Therefore we need to change the condition logged with number 1240 and execute the service for calling callback methods onStartCommand() and onUnbind().

```
    Method:<com.fx.a.FXS: int
onStartCommand(android.content.Intent,int,int)>:onStartCommand:com.fx.a.FXS:com.fx.a
    /*method entry*/
    /* code path starts*/
    entry
    <android.app.Service: int onStartCommand(android.content.Intent,int,int)>
    <com.fx.a.FXS: com.fx.a.g a()>
    <com.fx.a.g: int a(android.content.Intent,int,int)>
    Exit
    /* code path ends*/
```

```
/*method exit*/
Method:<com.fx.a.FXS: boolean onUnbind(android.content.Intent)>:onUnbind:com.fx.a.FXS:com.fx.a
/*method entry*/
/* code path starts*/
entry
<android.app.Service: boolean onUnbind(android.content.Intent)>
<com.fx.a.FXS: com.fx.a.g a()>
<com.fx.a.g: boolean b(android.content.Intent)>
Exit
/* code path starts*/
/*method exit*/

Method:<com.fx.a.g: void a(android.app.Service)>:a:com.fx.a.g:com.fx.a
/*method entry*/
/* code path starts*/
entry
<com.fx.a.d: java.lang.Object a(java.lang.Object,java.lang.reflect.Method,java.lang.Object[])>
Exit
/* code path ends*/
/* code path starts*/
<android.util.Log: int i(java.lang.String,java.lang.String)>
IFLOG:"INFO","replaced log information1240",
<com.fx.a.d: java.lang.reflect.Method a(java.lang.Object,java.lang.String)>
/* code path ends*/
/*method exit*/
```

Figure 7.2 Trace log for "Dowgin"

Step 3: Change if-conditions and extract APKs with changed ifs: In the next step, we need to change the if-conditions that we extracted in the previous step. In the Jimple representation of the code, conditions are for checking a statement and if it is not committed, it calls *goto* to direct the code to continue executing from an address as illustrated in Figure 7.3. Therefore, for directing our code to be executed following a given path, we need to change every condition in the mentioned path to False. We use Soot to make changes to the if-conditions (Figure 7.4).

Java code	Jimple code

```
public int onStartCommand(Intent
 intent, int flags, int startId) {

   TelephonyManager
      telephonyManager =
     (TelephonyManager)
      getSystemService(
      Context.TELEPHONY_SERVICE);
   String imei =
     telephonyManager.
     getDeviceId(); //source
   int zeroPos = 0;
   while (zeroPos < imei.length())
   {
```
	```
	public int onStartCommand(android.content.Intent, int, int)
	   {
	      de.ecspride.MainService $r0;
	      android.content.Intent $r1;
	      int $i0, $i1;
	      java.lang.Object $r2;
	      android.telephony.TelephonyManager $r3;
	      java.lang.String $r4;
	      char $c2;
	      $r0 := @this: de.ecspride.MainService;
	      $r1 := @parameter0: android.content.Intent;
	      $i0 := @parameter1: int;
	      $i1 := @parameter2: int;
	      $r2 = virtualinvoke $r0.<de.ecspride.MainService:
	```

```
      if (imei.charAt(zeroPos) ==
           '0')
         zeroPos++;
       else {
         zeroPos = 0;
         break;
       }
   }

   secret = telephonyManager.
     getSimSerialNumber();
     //source
     return 0;
   }
```

```
          java.lang.Object getSystemService
          (java.lang.String)>("phone");
     $r3 = (android.telephony.TelephonyManager) $r2;
     $r4 = virtualinvoke $r3.<android.telephony.
          TelephonyManager: java.lang.String getDeviceId()>();
     $i1 = 0;
   label1:
     $i0 = virtualinvoke $r4.<java.lang.String: int length()>();
     if $i1 < $i0 goto label3;
     staticinvoke <android.util.Log: int i(
        java.lang.String,java.lang.String)>("INFO", "replaced log
         information1");
   label2:
     $r4 = virtualinvoke $r3
        .<android.telephony.TelephonyManager: java.lang.String
        getSimSerialNumber()>();
     $r0.<de.ecspride.MainService: java.lang.String secret> = $r4;
     return 0;
   label3:
     $c2 = virtualinvoke $r4.<java.lang.String: char charAt
        (int)>($i1);
     if $c2 != 48 goto label4;
     staticinvoke <android.util.Log: int i
        (java.lang.String,java.lang.String)>("INFO", "replaced log
        information2");
     $i1 = $i1 + 1;
     goto label1;
   label4:
     goto label2;
   }
```

Figure 7.3 Jimple representation after adding if instrumentation

```
private static void changeIf(Body body, Unit ifUnit, PrintStream
                                   errorwriter){
   if(ifUnit instanceof soot.jimple.IfStmt){
        soot.jimple.IfStmt s = (soot.jimple.IfStmt) ifUnit;
        IntConstant zero = IntConstant.v(0);
        IntConstant one = IntConstant.v(1);
        soot.jimple.EqExpr compExpr=Jimple.v().newEqExpr(one, zero);
        //it is checking condition
        Unit generated = Jimple.v().newIfStmt(compExpr,s.getTarget());
        body.getUnits().insertAfter(generated, ifUnit);
        body.getUnits().remove(ifUnit);
   }
}
```

Figure 7.4 Code for changing if conditions in an app

7.1.2 Dynamic Analysis phase

We then execute the manipulated app and save the stack traces. By studying the traces, we can figure out the name of the method called by reflection call. We need to perform the static analysis again in order to extract the API calls in the method we obtained its name from studying the traces.

7.2 Case Study 1: Study on Genome Dataset

In the following case study, we assess the effectiveness of xHyDroid to extract API call traces from malicious apps. As discussed earlier, we focus on the service components of apps and this case study looks at malware families, which contain malicious operations in service components of Android app. Note that a malware may manifest itself in other components as well. This case study addresses the following research question:

RQ2: Can we use xHyDroid to extract the API call signatures of a malware?

7.2.1 Dataset

We use the Gnome repository of malicious repackaged Android applications provided by Zhou and Jiang (2012a). The dataset is publicly accessible on the web[10]. It contains over 1200 malware samples, categorized by malware families. We selected fourteen malware families that manifest their malicious operation mainly through the service components of an application and contain more than one sample in the dataset so that we can find the correct malicious service by comparing the service code in samples.

All selected malware samples read some of devices' confidential data such as DeviceID and IMEI number, geographical location, received SMS or Phone call data and send them to a remote server. While some families including GolDream (Symantec Report 2011a) and GPSSMSSpy (Yoshikawa 2012), NikySpy (Symantec Report 2011b), SndApps (Jiang 2011e) and Tapsnake (Symantec Report 2010) limited their malicious operations to hard coded leakage of such confidential data, some other families, namely, BgServ(Dominguez 2011), BeanBot (Xia 2015), DroidKungFu families (Jiang 2011a), GingerMaster (Jiang 2011b), Pjapps (Ballano 2011), and Plankton (Jiang 2011e) continue performing a variety of operations by getting commands from the server. Different commands in samples include downloading files, installing payload, blocking message, sending message to premium numbers, changing remote server address, sending GPS location, etc. HippoSMS (Jiang 2011c) was implemented to send message to premium numbers. While there are four samples in the Genome dataset, we carry out our experiment on only two samples, which use service component to host malicious operations.

For each malware family, the dataset contains applications that are obfuscated and others that are not. The obfuscated techniques vary from name alteration to the use of reflection and string encoding. To check the obfuscation used in applications, we retrieved the source code, using dex2jar to covert a dex file into Java. The distribution of obfuscated and non-obfuscated applications of each malware family is shown in Table 7.1.

[10]http://www.malgenomeproject.org/

Table 7.1 Distribution of obfuscated and non-obfuscation apps in each malware family in Genome dataset

Malware Family	**Num of apps**	**Used obfuscation**	**Num of non-obfuscated apps**
BgServ	9	9 sample with name alteration and reflection	0
BeanBot	8	8	0
DroidKungFu1	34	0	34
DroidKungFu2	30	0	30
DroidKungFu3	309	0	309
GingerMaster	4	4 sample with name alteration and reflection	0
GoldDream	47	6 sample with only name alteration and 13 sample with name alteration and reflection and encryption	28
GPSSMSSpy	6	0	6
HippoSMS	2	2 sample with name alteration	0
NickySpy	2	None	2
Pjapps	58	None	0
Plankton	11	11 sample with name alteration and reflection	0
SndApps	10	None	10
Tapsnake	2	None	2

7.2.2 Evaluation

To evaluate the effectiveness of xHyDroid in extracting API call signatures from the execution of the services infected by these families of malware we have performed the following steps:

Step1: We extracted the API call traces in methods through the lifecycle of a malicious service in malware app samples by using Soot,

Step 2: We extracted the API call traces using xHyDroid and,

Step 3: We compared the API call traces extracted by xHyDroid to the ones recorded in the first step.

We used Soot library to extract the call graph through the lifecycle of a service as described in Step 1. We extract the call graph rooted from methods named as the starting point. Since we want to extract the API calls though a service lifecycle, we assigned callback methods in a service as staring points; including, onCreate(), onStart(), onDestroy(), onBind(). If onBind() is present, it is necessary to add public methods since they can be called directly like some samples in DroidKungFu2.

Since Soot can not include the methods called by reflection in the call graph, we study the java code of the apps where a reflection call is present. Apps that are obfuscated using name alterations and reflection, while passed string parameters that shows the name of the called method, are not encrypted, which allowed us to use them as part of the testing set. We extract the name of these methods and add them in the call graph. Note that, here, we are trying to evaluate the xHyDroid and the result of Step 1 is our baseline.

For each method in the call graph of a service, we extracted the post dominator of all API calls through the control flow of each method. Post-dominators record the sequence of APIs called successively through code paths in a method.

In Step 2, we extracted the API call traces using xHyDroid. To run xHyDroid, we needed to exercise the obfuscated apps (the testing set) with different values of CondParameter conditions to be able to cover most execution paths. For each application, we started by setting all CondParameter conditions to false and extracting the API calls that would result from executing the application. In the second execution phase, each condition was successively set to True while the remainder was set to False. We continued setting the CondParameter conditions until we did not see any changes to the extracted API calls. We applied this process to the applications of each malware family.

In Step 3, to evaluate whether the API call trace extracted by xHyDroid is present in API call traces extracted in Step 1, we looked for the longest sequence of API calls presents in the trace and in a post-dominator of the starting point method. Then we removed that sequence from the trace and continued the comparison of the remaining APIs in the trace in the post-dominators of successor methods of the starting point method in the call graph. Our algorithm recursively continued such similar comparison for the reminder of API calls in the successor methods in the call graph and it’s post-dominator.

To simplify and limit the computation time needed for trace matching, we applied two optimizations in our algorithm. First, for methods from the package “java.lang.String”, we only recorded the name of the package, rather than the name of the specific method. This method frequently occurs in our sample, so recording only the package mane, rather than the specific method, results in a substantial optimisation. Note that this package does not require any permission to be used in Android apps. Moreover, if there are repeated consecutive calls of methods from this package in a post-dominator, we only log them once. Another optimization was recording only the post-dominators that contain API calls.

7.2.3 Results

We were successful in extracting API calls using xHyDroid in all of the samples in Genome dataset. Table 7.2 shows the number of distinct API calls for each malware family. All the samples in these malware families have a single malicious service, except NikySPy which has seven malicious services. In total, there were 250 distinct API calls in these services as are written in the table.

Table 7.2 Number of distinct API calls for each malware family

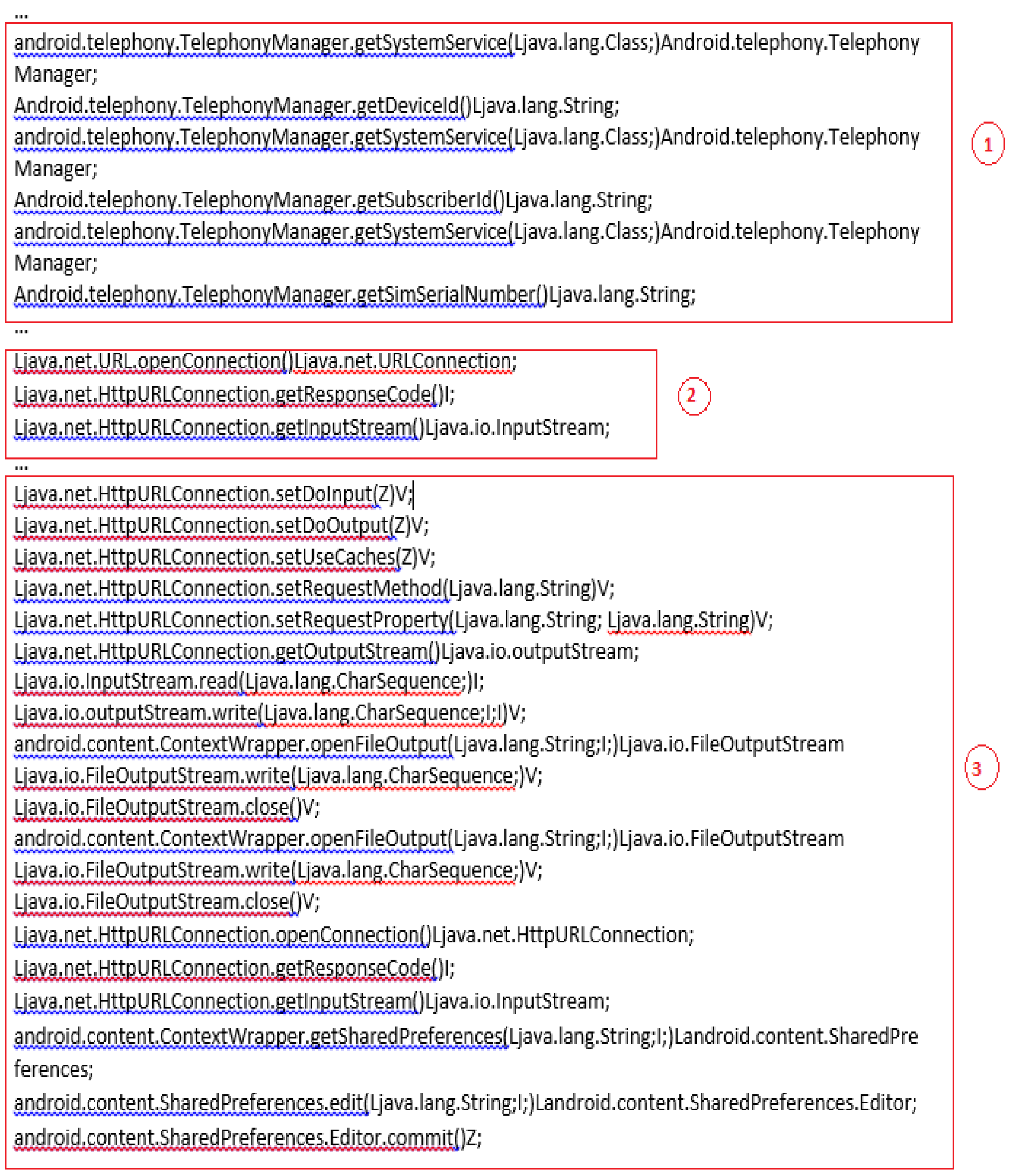
No.	Malware Family Name	Number of distinct API calls
1	BgServ	106
2	BeanBot	29
3	DroidKungFu1	104
4	DroidKungFu2	82
5	DroidKungFu3	116
6	GingerMaster	70
7	GoldDream	72
8	GPSSMSSpy	45
9	HippoSMS	27
10	NickySpy	251*
11	Pjapps	105
12	Plankton	71
13	SndApps	44

14	TapsnaKe	22

```
{entry,<com.GoldDream.zj.zjService:boolean
    fileIsExists(java.lang.String)>,exit}
{<com.GoldDream.zj.zjService:java.lang.String
    getKeyNode(java.lang.String,java.lang.String)>,
<java.lang.Integer: java.lang.Integer valueOf(java.lang.String)>,
<java.lang.Integer: int intValue()>,
<java.io.FileInputStream: void <init>(java.lang.String)>,
<java.io.FileInputStream: int available()>,
<java.io.FileInputStream: void close()>, exit}
{<java.lang.String: int lastIndexOf(java.lang.String)>,
<java.lang.String: int length()>,
<java.lang.String: java.lang.String substring(int,int)>,
<android.content.ContextWrapper: java.io.FileOutputStream
    openFileOutput(java.lang.String,int)>,
<java.lang.String: void <init>(java.lang.String)>,
<java.lang.String: byte[] getBytes()>,
<java.io.FileOutputStream: void write(byte[])>,
<java.io.FileOutputStream: void close()>, exit}
```

Figure 7.5 Post-dominator of method “ void com.GoldDream.zj.zjService.CheckAndClearFile(java.lang.String)” in GoldDream

To exemplify, Figure 7.5 shows the post-dominators for the method CheckAndClearFile() in GoldDream. Based on post-dominators, all the APIs inside a curvy bracket are to be called consecutively before the method exits. In Figure 5, a sample trace of API calls extracted by xHyDroid is shown. For GoldDream, as shown in Figure 7.6, the malware reads confidential data such as device ID and IMEI (see section marked (1)), and establishes a connection with a remote server (shown in (2)). It then reads the file containing recorded message and phone call information by a broadcast receiver component and sends its content to the remote server (shown in (3)).



```
...
android.telephony.TelephonyManager.getSystemService(Ljava.lang.Class;)Android.telephony.Telephony
Manager;
Android.telephony.TelephonyManager.getDeviceId()Ljava.lang.String;
android.telephony.TelephonyManager.getSystemService(Ljava.lang.Class;)Android.telephony.Telephony
Manager;
Android.telephony.TelephonyManager.getSubscriberId()Ljava.lang.String;
android.telephony.TelephonyManager.getSystemService(Ljava.lang.Class;)Android.telephony.Telephony
Manager;
Android.telephony.TelephonyManager.getSimSerialNumber()Ljava.lang.String;
...
Ljava.net.URL.openConnection()Ljava.net.URLConnection;
Ljava.net.HttpURLConnection.getResponseCode()I;
Ljava.net.HttpURLConnection.getInputStream()Ljava.io.InputStream;
...
Ljava.net.HttpURLConnection.setDoInput(Z)V;
Ljava.net.HttpURLConnection.setDoOutput(Z)V;
Ljava.net.HttpURLConnection.setUseCaches(Z)V;
Ljava.net.HttpURLConnection.setRequestMethod(Ljava.lang.String)V;
Ljava.net.HttpURLConnection.setRequestProperty(Ljava.lang.String; Ljava.lang.String)V;
Ljava.net.HttpURLConnection.getOutputStream()Ljava.io.outputStream;
Ljava.io.InputStream.read(Ljava.lang.CharSequence;)I;
Ljava.io.outputStream.write(Ljava.lang.CharSequence;I;I)V;
android.content.ContextWrapper.openFileOutput(Ljava.lang.String;I;)Ljava.io.FileOutputStream
Ljava.io.FileOutputStream.write(Ljava.lang.CharSequence;)V;
Ljava.io.FileOutputStream.close()V;
android.content.ContextWrapper.openFileOutput(Ljava.lang.String;I;)Ljava.io.FileOutputStream
Ljava.io.FileOutputStream.write(Ljava.lang.CharSequence;)V;
Ljava.io.FileOutputStream.close()V;
Ljava.net.HttpURLConnection.openConnection()Ljava.net.HttpURLConnection;
Ljava.net.HttpURLConnection.getResponseCode()I;
Ljava.net.HttpURLConnection.getInputStream()Ljava.io.InputStream;
android.content.ContextWrapper.getSharedPreferences(Ljava.lang.String;I;)Landroid.content.SharedPre
ferences;
android.content.SharedPreferences.edit(Ljava.lang.String;I;)Landroid.content.SharedPreferences.Editor;
android.content.SharedPreferences.Editor.commit()Z;
```

Figure 7.6 A sample API call trace of GoldDream

It is worth noting that the analysis of services will reveal only part of the malicious operations since the malware can also involve other components of the application such as the Broadcast Receiver components. We intend to extend this research to cover other Android app components.

7.3 Case Study 2: Study on AndroZoo Dataset

7.3.1 Dataset

AndroZoo (Li et al. 2017a) contains a list of pairs of original and repackaged apps. AndroZoo contains 15, 278 repackaged versions of 2, 777 original apps, organized in pairs. One original app may have multiple repackaged versions. Moreover, all of the original apps in the dataset are benign but each repackaged app is designated as malware by at least one antivirus.

We used the tool ApkTool to extract the name of every service in each app. By comparing the services that exists in each original app and the services present in its repackaged pair, we were able to identify the services added in repackaged apps that contain malware. Since the repackaged app is a clone of original app into which malicious code has been embedded, we can reasonably consider the added services in the repackaged apps as malicious services.

Although there may be some misidentification where the added service is not malicious, there were no other options for us to identify whether or not a service in an app is specifically malicious or not. Moreover, there may be a service in a repackaged app, which also exists in the original app, but was modified to embed malicious operations. Detecting these service components requires a manual analysis of the code, which is impractical considering the large number of services in a dataset as large as that of AndroZoo. Therefore, we have limited our study to the services that only exist in repackaged versions of benign apps. We have found 141 such malicious services.

7.3.2 Results and discussions

Li et al. (2016) have grouped the legitimate use of reflection by Android developers in four main areas as follows:

1. Developers use reflection to implement generic functionality. For example, reflection is used to produce the initialization of Collection as List or Set based on the passed parameters' type of the method in runtime.

2. Developers use reflection for maintaining backward compatibility. For example, a program may check the targetSdkKversion of a device and call the APIs available for that Android version. This usage is common among Android developers. It is also the recommended way on Android developers' official site (Android Developer Documentation 2018) for ensuring backward compatibility for different devices, and SDK versions.

3. Developers protect their apps against reverse engineering by separating the core functionality of their app in a library and loading it at runtime. Thus, reflection is used in the app for calling the methods of the library.
4. Developers sometimes use the reflection to access internal APIs, which are supposedly for the usage of system apps. For example, getService() of class ServiceManager is written to be used by system apps. However, a developer can use reflection call to reach that method at run time.

We conducted an exploratory study over services of AndroZoo in Chapter 5. After performing Phase1, we reconfirmed the number of services having reflection calls as listed in Table 5.2 in Chapter 5. We determined that 14 out of 24 malware families (108 out of 141 malware samples) have reflection calls in their services. Therefore, for these families we need xHydroid to extract the API call traces. For the remainder of the ten malware families, 33 out of 141 samples, we used only static analysis to extract the API call traces.

Out of the 108 malware samples that used reflection, 32 did so in order to access internal APIs (as presented in bullet 4 above), including samples in “Domob”, “Youmi”, “Adwo” and “Waps” malware. For example in Domob, a service named 'com.goldsoft.game.tuoyu.EmulatorService' contains a reflection without encryption. Based on the actions sent by intent to the service, this call uses system notification to execute in either the background or the foreground. The actions are either com.androidemu.actions.FOREGROUND, or com.androidemu.actions.BACKGROUND. The part of the code in Domob using reflection call is shown in Chapter 5 Figure 5.4. In total, there are 11 samples in our dataset using reflection for this purpose.

The other use of reflection in our malware samples is related to calling reflection calls in order to use method in libraries loaded at run time (as presented in Bullet 3). In malware families such as “Shixit” as shown in Figure 5.7 of Chapter 5, by using xHyDroid we were able to extract the name of the methods called but as the method were in a class in a loaded library, we were not able to perform static analysis on those libraries and extract the API calls. However, in some samples the reflection is used to hide the name of a method that already exists in the code, such as in some samples of Dowgin. In total, we were able to extract the API calls in 21 malicious services, which used reflection for hiding the called method.

7.4 Threats to Validity

A threat to validity exists in the way we assessed the results of xHyDroid. We used a combination of source code analysis and trace inspection. We mitigated this threat by carefully examining each segment of a trace and comparing it to the API calls in the code, as well as to the description of these malwares provided in the Gnome documentation.

A threat to external validity exists in the generalization of our approach to other datasets. We only evaluated our approach on the service components of malware, and further experiments to generalize the approach to cases where malicious operations are added in other components are left as a future work.

7.5 Conclusion and Future Work

In this chapter, we extracted the API call traces in malicious services from the Genome and AndroZoo dataset. We evaluated the correctness of the detection of reflected calls by xHyDroid by studying the decompiled java code of apps. The main drawback of this evaluation procedure relates to human error. To mitigate this risk, different collaborators verified the results. We also tried to use DroidRA (Li et al. 2016) to extract the name of the methods in reflected calls and compare them with our results. Unfortunately, DroidRA could not be run on most of the malware samples.

Chapter 8 Detection of Malicious Services Using API Call Traces

8.1 Introduction

In Chapter 5, we observed that malware differs from benign code in that it is much more loosely connected with the rest of the app's code than benign classes. Since the connection with other components is often performed through calling dedicated APIs methods, the distinction between malware and benign code could be educed from an examination of the API calls in service components. Likewise, we showed in Chapter 5 that malware attempts to obfuscate its presence, again often through the use of dedicated API calls, such as Schedule and Reflection.

Consequently, in this chapter, we present a novel approach for malware detection based upon the profile of API calls preponderantly present in both malware as well as in benign services. For this purpose, we will apply machine learning algorithms over the feature set obtained from API calls through the lifecycle of service components of apps. In fact, we utilize the observations of Chapter 5 to model the behavior of benign services as well that of malicious services.

As explained in Chapter 3, there exist research studies for classifying malware and benign apps using machine learning algorithm over the API call traces. Considering the fact that benign apps differ from each other since they are developed for a variety of functionalities, learning these apps' behaviour is impractical. Therefore, researchers have proposed approaches to learn the characteristic malicious behaviour of malware. These detection mechanisms are based on API calls extracted from malware. As we explained in Chapter 4, none of those approaches evaluate their work using a dataset that contains pairs of benign and repackaged malware. Since repackaged apps are very similar to benign apps, a detection approach may be widely impacted if the evaluation set contains both benign apps and their repackaged version.

In this chapter, we perform detection of malware by studying the service components of apps. We train the machine learning models using the API calls that preponderantly exist in benign services as well as malicious ones.

8.2 Classification Approach

Our approach consists of training a classifier using API calls as shown in Figure 8.1. There are two classes, malware and benign apps (while we focus specifically on repackaged apps with embedded malicious operations, we will use the general term malware in the remainder of this chapter for the convenience of the reader). It should be noted that other machine learning paradigms could also be used such as anomaly detection techniques, which rely on one-class classification (Khreich et al. 2017) –training on normal behavior and testing on malicious behavior.

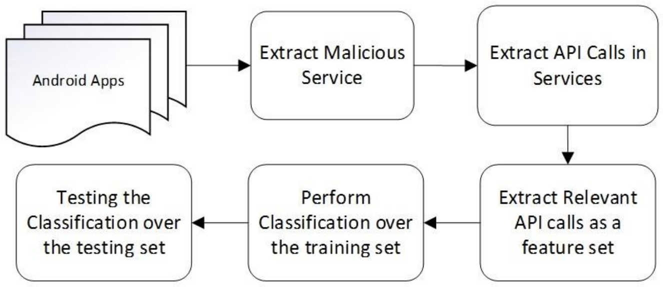

Android Apps
Extract Malicious Service
Extract API Calls in Services
Extract Relevant API calls as a feature set
Perform Classification over the training set
Testing the Classification over the testing set

Figure 8.1 Overview of the classification approach

The approach proceeds through the following steps:

Extract Malicious Services. We first need to extract the malicious services that will be used for classification. We obtained the names of the services present in each app from the Androidmanifest.xml file. In the Genome dataset, we used the code similarity of malware samples in the same malware family in order to identify malicious services. In the AndroZoo dataset, we identified malicious services by singling out the services added in repackaged applications in comparison to the original benign application. This procedure is detailed in Section 5.

Extract API calls. In the second step, we extracted the lifecycle of API calls for each service component. As explained in chapter 3, the lifecycle is simply the set of ordered sequences of callback methods (such as *onStart*) that can be called by Android system through the execution of the service. We first extract the method call graph of a service component. It is obtained through a static analysis of each app's service components using Flowdroid (Arzt et al. 2014) in the Soot framework (Bartel et al. 2012). We assigned the callback methods in each service, such as *onCreate(), onStart(), onDestroy()* or *onBind(),* as entry points. Then, we went through the call graph and extracted the API calls from the methods. Since we sought to extract the API calls of a service, if the method *onBind()* is implemented in an application, it was necessary to add the public methods in that service component as entry points as well because they can be called directly by other components. Note besides the callback methods in a service lifecycle, there is a set of callback methods that are common to all application components including *onConfigurationChanged()* and *onLowMemory()*. It is necessary to consider them as possible entry points to the call graph of a service as well.

Feature Selection. Using every extracted API call as our feature set for classification increases the number of false positives since these calls may exist in benign applications as well as in malware samples. To improve the accuracy, Aafer et al. (2013) used the most frequent API calls as features, extracting them from each of the components of an app. In our case, we considered a different metric: the most relevant API calls that distinguish the class to which an app belongs. In other words, we focused on the API calls that are frequently present in malware services but not in benign services and vice versa. For example, some API calls such as *android.telephony.SmsManager.sendTextMessage()* (used to send sms messages to premium rate numbers without the user's consent) frequently occur in malicious services while others, such as *android.app.NotificationManager()* (used to send notifications to the app's user) are frequently used in the services of benign application.

To identify such API calls, we applied Pearson's correlation coefficient. We used the WEKA library for feature selection, which computes the correlation between each feature and the output class and selects only the top *N*% of those features that have a moderate to-high positive or negative correlation (e.g., the top 25% closest to −1 or 1). We repeated each experiment with the top 75%, 50%, 25% and 12.5% most relevant API calls as the feature set. The results improved for feature set from 75% to 50% and from 50% to 25%, but results for the top 12.5% were not markedly different than those for 25%. In this work, we present the results obtained by using the top 25% most relevant API calls.

Classification. We retained 30% of the data, both malware and benign, as a testing set and used the remaining 70% of the data as a training set. The dataset was divided into two classes, one for malware services and one for benign services. We tested our approach using three classification techniques, namely Decision Tree, Random Forest, and linear Support Vector Machines (SVM). Although other algorithms could have also been employed, we chose these algorithms because they have been widely used in similar studies including the one by Aafer et al. (2013). We used 10-fold cross validation (Kohavi 1995) to measure accuracy and to fine-tune the models.

Unbalanced data. Moreover, since the number of benign samples is larger than the number of malware samples, we used SMOTE (Chawla et al. 2002) to cope with the imbalanced dataset. Using SMOTE, we over-sampled the minority class, labeled "malware" by creating "synthetic" examples. SMOTE creates new examples by taking into account the *k* nearest minority class neighbors. Chawla et al. (2002) showed that SMOTE provides better results in comparison to over-sampling that only replaces minority class (the class with fewer samples than the other class) examples when the cost of classifying an abnormal example as a normal example is much higher than the cost of the reverse error. This applies to our case where we favor the detection of malicious services over reducing false negatives. In our experiments, we used the default value for *k*, namely 5, and chose 17 times oversampling.

8.3 Experiment Setup

8.3.1 Datasets

We used the following resources to obtain benign and malware applications for our experiments. The size of the dataset used in this study is detailed in Table 8.1.

GooglePlayStore dataset: Li et al. (2017b) collected one of the largest datasets of Android apps from various markets, including the official Google Play Store. We randomly downloaded 655 benign apps from their dataset. These apps were published in Google Play Store, a generally trusted resource. Each of these apps was scanned by VirusTotal (2018) and classified as benign.

Genome dataset: The Genome repository was developed by Zhou and Jiang (2012a). It contains 1226 malware apps categorized in 49 families of malware. This repository includes malwares dating back from 2012. Based on code similarity of services in a variety of samples in each of the malware families, we identified the malicious services in each malware family. In this regard, we retrieved the source code from the samples by using dex2jar[11] which converts a dex file into a Java source file. To increase our confidence in the results, we also studied the malware analysis reports provided by Zhou and Jiang (2012b). In sum, we found 67 malicious services in the malware apps of the Genome dataset.

AndroZoo dataset.: AndroZoo (Li et al. 2017b) contains a list of pairs of original and repackaged apps. AndroZoo contains 15,278 repackaged versions of 2,777 original apps, organized in pairs. One original app may have multiple repackaged versions. Moreover, all of the original apps in the dataset are benign but each repackaged app is recognized as malware by at least one antivirus. We used the tool AndroGuard (Desnos 2015) to extract the name of every service in each app. By comparing the services that exist in each original app and the services present in its repackaged pair, we were able to identify the services added in repackaged apps that contain malware. Since the repackaged app is a clone of original app into which malicious code has been embedded, we can reasonably consider the added services in the repackaged apps as malicious services. Although there may be some misidentification where the added service is not malicious, there were no other options for us to identify whether or not a service in an app is specifically malicious or not. Moreover, there may be a service in a repackaged app, which also exist in the original, but was modified to embed malicious operations. Detecting these service components requires a manual analysis of the code, which is impractical considering the large number of services a dataset as large as that of AndroZoo. Therefore, we have limited our study to the services that only added in repackaged version of benign apps. There are 141 added services in repackaged malware. But, as explained in Chapter 7, we were able to extract the API calls in 65 malicious services.

[11] https://github.com/pxb1988/dex2jar/wiki.

Table 8.1 Distribution of training and testing datasets

Services	Training	Testing	Total
Benign	753(70%)	323(30%)	1076(100%)
Genome	47(70%)	20(30%)	67(100%)
AndroZoo	46(70%)	19(30%)	65(100%)

8.3.2 Evaluation Metrics

To evaluate the performance of our approach, we use precision, recall, and F1-measure. Precision indicates how many samples selected as malware are truly malware. Recall is the percentage of malware samples correctly detected as such by machine learning.

$$Recall = \frac{Number\ of\ apps\ correctly\ detected\ as\ malware}{Total\ number\ of\ malware\ in\ the\ dataset}$$

$$Precision = \frac{Number\ of\ apps\ correctly\ detected\ as\ malware}{Total\ number\ of\ samples\ detected\ as\ malware}$$

F1-measure (F1 score) is used to show the harmonic mean of precision and recall where its best value is 1 and its worst value is 0. It is defined by the following formula:

$$F\text{-}meature = 2 * \frac{Precision * Recall}{Precision + Recall}$$

8.4 Experiments with Various Feature Sets

We conducted some sets of experiments. In the first Experiment, we extracted feature sets from malicious services only. In the rest of the experiments, we added the relevant API calls obtained from benign applications. The objective is to understand whether knowledge of API calls that appear in malware alone is sufficient to classify app services or if there is a need to include knowledge of API calls used in benign apps as well. We also studied how the result will change by training the classification models with recent set of malware.

Experiment 1: Feature Set Extracted from Genome Malicious Services only. In this experiment, we trained the classification algorithms using relevant API calls extracted from the service components of the Genome malware samples in the Genome training set. As discussed above, we tested three classification algorithms: SVM, Random Forest, and Decision Tree.

Experiment 2: Feature Set Extracted from Genome Malicious Services and Benign Services. In this experiment, we sought to determine if using the relevant API calls in benign services, in addition to the ones from extracted malware services, increases the performance of the detection scheme. In this respect, we applied the classification over API calls extracted from both malware and benign services. Therefore, this feature set also includes the API calls that are frequently used in benign services but not in malware services. Similar to experiment 1, the services in the Genome and benign datasets were divided into two sets with 30% of them reserved for training and the remainder used for testing.

Experiment 3: Training Set Extracted from Genome Malicious Services and Benign Services, Testing Set from Both Genome and AndroZoo. In this experiment, as was the case in experiment 2, we used the Genome dataset for training, but for testing, we additionally included services obtained from AndroZoo. Since the Genome dataset was collected in 2012, it does not contain recent malware. We used the AndroZoo dataset to obtain more recently developed malicious services. This experiment sought to determine how well the trained model would work in detecting the newer malware.

Experiment 4: Training Set and Testing Set Extracted from Genome and AndroZoo Malicious Services and Benign Services. In a final experiment, we used a training set that includes relevant API calls from malicious services in the Genome dataset and the AndroZoo dataset as well as relevant API calls from benign services. The testing set includes 30% of all malware services in the Genome and AndroZoo datasets and 30% of benign services. Accordingly, the remaining 70% of the apps were used as a training set.

8.5 Results and Discussions

The results of classification over these datasets, for experiment 1 and experiment 2, for each of the three algorithms we tested, is shown in Table 8.2.

The results indicate that using the API calls from benign services as well as the API calls from malicious services provides for a better classification. Optimal results are obtained when we using Random Forest as the classification algorithm. These results also confirm that experiment 2 provides optimal accuracy.

Table 8.2 Results of experiments 1 (top) and 2 (bottom) using 70% of dataset as training set and 30% of dataset as testing set

Classification Algorithm	Precision	Recall	F-measure
SVM	0.75	0.6666	0.7058
Decision Tree	0.9166	0.6111	0.73333
Random Forest	0.9411	0.8888	0.9142
SVM	0.9473	1.0	0.9729
Decision Tree	0.8421	0.8888	0.8648
Random Forest	0.9444	0.9444	0.9444

The results of our third experiment, in which we leveraged AndroZoo to obtain a more recent dataset, are shown in Table 8.3 (top). They show that the accuracy has actually decreased, suggesting that the behavior of malware has changed in the new malware samples.

Table 8.3 Results for experiment 3 (top) and 4 (bottom)

Classification Algorithm	Precision	Recall	F-measure
SVM	0.2142	0.9322	0.3483
Decision Tree	0.2132	0.9312	0.3469
Random Forest	0.2142	0.9322	0.3483
SVM	0.9827	0.8888	0.9333
Decision Tree	0.9827	0.8888	0.9333
Random Forest	0.9827	0.8888	0.9333

The results of our final experiment are given in Table 8.3. The results were similar for each of the classification algorithms.

To better understand how the service's behavior differs in malware in comparison to benign apps, we examined the top 30 most relevant APIs, identified using the Pearson Correlation as the feature set in Experiment 1, 2 and 4. Note that feature set in Experiment 2 and 3 are similar and their testing set are different. Since Experiment 1 uses only API calls from malicious services, most of the top API calls are related to reading confidential information or accessing resources such as *android.content.ContextWrapper.getFilesDir, android.telephony.TelephonyManager.getLine1Number* or *android.telephony. SmsManager.sendTextMessage*. In Experiment 2, since the feature set includes benign services, there are obvious differences in the top 30 APIs. In this case, API calls come from benign services in the feature set such as *android.content.Intent. getAction*, *java.util.Iterator.next* or *org.json.JSONObject. toString*. In Experiment 3, the malicious services also contain services from the AndroZoo dataset, and the feature set in Experiment 3 is different from the one in experiment 2, despite the fact that both use a mix of benign and malicious services. Several API calls indicate that benign services communicate with the rest of the code, or with the app's user such as *android.app.NotificationManager,* which is designed to show notifications to the user.

The two studies that are closest to our work are DroidAPIMiner (Aafer et al. 2013) and MaMaDroid (Mariconti et al. 2017). DroidAPIMiner used frequent API calls of malware samples as feature sets and MaMaDroid used the sequence of API calls. We used the relevant API calls, which include API calls that are frequent in malware but not frequent in benign apps. Mariconti et al. (2017) showed that MaMaDroid outperforms DroidAPIMiner. Our results are very similar to the results obtained by MaMaDroid (similar f-measure of 99%). Moreover, since we directly study the service's API calls, our approach can detect the malicious behavior even if a call is hidden through reflection. A static analysis that simply goes through all code paths would not detect malicious behavior in a service that is called through reflection.

8.6 Threats to validity

The main threat to validity lies in the way in which we identified the malicious services. We achieved this by comparing the code of samples of each of the malware families and the description of the malware. This approach presents a risk of human error. To mitigate this risk, we asked different collaborators to verify the results.

Furthermore, the benign applications that we used in our experiments are all free. We have not experimented with paid applications. Further experimentation with paid apps is left for future work.

Table 8.4 Top 30 relevent API selected as feature set in various experiments

Top 30 Relevant API Calls		
Experiment 1	Experiment 2	Experiment 4
{android.content.ContextWrapper.getSystemService}	{android.content.ContextWrapper.getSystemService}	{java.lang.Object}
{android.content.ContextWrapper.getApplicationContext}	{android.app.Service.<init>}	{android.content.Intent}
{android.content.ContextWrapper.getContentResolver}	{android.app.IntentService.<init>}	{java.util.ArrayList}
{android.telephony.SmsManager.sendTextMessage}	{java.lang.IllegalArgumentException.<init>}	{android.content.Context}
{android.content.ContextWrapper.stopService}	{android.content.ContextWrapper.getSharedPreferences}	{java.lang.System}
{android.telephony.SmsManager.getDefault}	{android.content.ContextWrapper.getApplicationContext}	{android.app.Service}
{android.content.ContextWrapper.startService}	{android.content.Context.getApplicationContext}	{android.os.AsyncTask}
{android.content.ContextWrapper.getPackageManager}	{java.lang.IllegalStateException.<init>}	{android.content.res.Resources}
{android.telephony.gsm.SmsManager.getDefault}	{java.util.Iterator.hasNext}	{android.app.Activity}
{android.telephony.gsm.SmsManager.sendTextMessage}	{java.util.Iterator.next}	{java.lang.Thread}
{android.content.ContextWrapper.getFilesDir}	{java.util.HashMap.<init>}	{android.app.PendingIntent}
{android.telephony.TelephonyManager.getLine1Number}	{java.util.List.iterator}	{android.os.Handler}
{android.content.ContextWrapper.registerReceiver}	{java.util.ArrayList.<init>}	{java.net.HttpURLConnection}
{android.telephony.SmsMessage.getMessageBody}	{java.util.Map.put}	{android.content.pm.PackageManager}
{java.io.InputStream.available}	{java.util.Map.get}	{java.net.URL}
{android.database.ContentObserver.onChange}	{android.os.Handler.<init>}	{java.util.List}
{java.net.Socket.getInputStream}	{java.util.HashSet.<init>}	{android.widget.ImageView}
{java.net.Socket.connect}	{android.content.Intent.getStringExtra}	{java.io.File}
{java.lang.Process.getOutputStream}	{java.util.Set.iterator}	{android.content.SharedPreferences}
{android.telephony.SmsMessage.createFromPdu}	{android.content.Context.getPackageName}	{android.app.Notification}
{java.io.DataOutputStream.writeBytes}	{java.lang.Class.getName}	{android.app.NotificationManager}
{java.text.SimpleDateFormat.applyPattern}	{android.os.Looper.getMainLooper}	{java.util.Iterator}
{android.net.NetworkInfo.getExtraInfo}	{org.json.JSONObject.<init>}	{android.graphics.Bitmap}
{android.content.BroadcastReceiver.abortBroadcast}	{java.lang.NullPointerException.<init>}	{android.os.Bundle}
{android.content.ContextWrapper.getApplicationInfo}	{java.util.ArrayList.add}	{java.lang.Exception}
{android.content.pm.PackageItemInfo.loadLabel}	{java.util.List.size}	{java.util.HashMap}
{java.net.Socket.isConnected}	{android.content.ContextWrapper.getContentResolver}	{android.database.sqlite.SQLiteDatabase}
{android.net.wifi.WifiManager.getWifiState}	{android.content.Intent.getAction}	{android.database.Cursor}
{java.util.Hashtable.containsKey}	{java.util.Map.keySet}	{java.io.InputStream}
{java.lang.Process.waitFor}	{java.lang.Object.getClass}	{java.io.FileOutputStream}
{android.net.wifi.WifiManager.setWifiEnabled}	{java.util.List.get}	{android.content.SharedPreferences$Editor}
	{java.util.Map.containsKey}	{java.util.Set}
	{java.util.List.add}	{java.lang.Class}
	{android.text.TextUtils.isEmpty}	{java.lang.Math}
	{android.app.PendingIntent.getActivity}	{java.lang.NullPointerException}
	{android.content.Intent.putExtra}	{java.util.Date}
	{android.content.pm.PackageManager.getPackageInfo}	{android.graphics.BitmapFactory}
	{java.util.Map.remove}	{android.widget.Toast}
	{org.json.JSONObject.toString}	{android.widget.RelativeLayout}
	{java.lang.Integer.intValue}	{android.os.Environment}
	{java.util.HashMap.put}	{org.apache.http.HttpResponse}
		{org.apache.http.impl.client.DefaultHttpClient}
		{java.util.Random}
		{android.app.ProgressDialog}
		{android.net.NetworkInfo}
		{org.apache.http.StatusLine}
		{android.net.ConnectivityManager}

Chapter 9 Conclusion and Future Work

9.1 Summary of Findings and Contributions

In this thesis, we performed an empirical study on repackaged apps and addressed five research questions (Chapter 4). The summary of findings is as follows:

- Repackaging is a common way to distribute malware even in trusted stores such as Google Play Store, as proven by the considerable number of repackaged apps and their increasing trend over time as apps in AndroZoo dataset.
- Repackaged apps are widely used to spread a specific form of malware, namely adware.
- Adware samples behave different from Trojan samples. Permissions frequently used in adware are different from those in Trojan. Moreover, there are some Adware, which does not require additional permissions to execute. In addition, the frequency of APIs called in adware is similar to that in Trojan. Therefore, adwaredetection cannot be made based on API calls alone.
- Apps that do not use name changing obfuscation techniques are more likely to be repackaged.
- Repackaged apps have high start rate and number of downloads. It suggests that repackaged malware was successful in hiding their malicious behavior from the apps' user.
- Number of components, name of components, permissions in repackaged apps are mostly similar to those in the original pairs.

Base on the findings in an empirical study of repackaged apps, the name of components is left unchanged after repackaging. We used this finding to propose an indexing scheme to record the apps based on the hash of activity components name of apps (Chapter 4).

We studied the malicious service behaviours in malware families and present the findings in Chapter 5. The findings show that malicious services, contrary to benign ones, tend to do malicious operations in the background and have loose connection with rest of the code. according to these findings we focus on detecting malware by studying their services.

We were required to extract the API calls in order to utilize them as the feature set of classification approach. We proposed a hybrid approach using static and dynamic analysis to extract the API calls through the lifecycle of services (Chapters 6 and 7). Finally, we used a list of preponderant API calls in benign and malicious services to classify them (Chapter 8) with the objective of detecting repackaged apps.

9.2 Opportunities for Future Work

Adware detection: In our empirical study on repackaged apps, we showed that a large number of malware apps are in fact a type of adware. We also showed that they have very similar features in comparison to the original benign apps. Therefore, detecting adware becomes more difficult because of these similarities. On the other hand, in adware it is not only the end user and developer who suffer; the ad provider also loses revenue by paying the adware developer for advertisements shown through the adware. Studying adware and proposing targeted detection approaches is very promising.

Evaluating detection approaches: Previous malware detection approaches evaluate their methods in a dataset of benign and malware apps. We showed in Chapter 4 that repackaged apps have similarities with their original pair. Especially for adware samples, API call frequencies are also the very similar. Therefore, we suggest to re-evaluate the previously proposed approaches in a dataset containing pair of repackaged malware and their original benign apps. It would show the results using a dataset that are very near the real world of existing apps.

Protection against repackaging through obfuscation: Detection of repackaged apps, as shown in this thesis, leads to detecting malware; but it will not shield the the original app developer from the associated revenue loss. As shown in the findings in Chapter 4, original apps (that get repackaged) mostly do not have name-changing obfuscation. It suggests that the repackaged app developers find such apps easier to manipulate. Previous research (Zhou et al. 2014, Luo et al.

2016, Zeng et al. 2018, Zeng et al. 2019, Tanner et al. 2019 and He et al. 2019) proposed the use of obfuscation for protecting the apps. We suggest that by using the findings in Chapter 4, in addition to studying obfuscation techniques, , it may be possible to devise new approaches for protecting apps against repackaging.

Code Authentication: Some studies aim to find ways to protect the apps by checking whether the code is changed while the app executes (e.g., (Zhou et al. 2013a), (Ren et al. 2014), (Zhou et al. 2014) and (Lee et al. 2014)). Researchers proposed recording certain features of each app and comparing them while the app is executing. The main limitation of these approaches is that it requires the use of a specially manipulated DVM. Other research studies use watermarking and connecting the watermarking to the original code. They changed the code of the app in a way that it stops functioning if the presumed user input is not obtained by the app. Therefore, the manipulation of code can be detected by the end user. These approaches are not practical as they require that changes be made to the DVM or rely on the detection by the end user who is not generally knowledgeable in this area. Since most malware are adware and ad providers also lose revenue because of repackaging, we suggest proposing an approach to authenticate the code by ad networks. Ad networks provide SDK for developers to query advertisements. They also record the developer's identity for payment purposes. Therefore, it seems logical that they will be motivated to authenticate the app's request for advertisement. Further studies are needed to understand the ad SDKs and the applicability of authenticating apps by ad networks.

Usage of HyDroid: We used HyDroid to extract API calls in the service components. There are possibilities to use HyDroid for security analysis as well. For example, Pan et al. (2017) studied the emulator detection approaches used by malware. When an app contain emulator detection code, debugging the code in an emulator can be hard or even impossible. Using HyDroid to change the parts of the code related to emulator detection can be an interesting avenue for future work.

Studying other components of apps and including temporal order of API call traces: In this thesis, we studied the behavior of services and the differences between malware and benign services. In fact, we studied a common location for embedding malicious operations in malware samples. We suggest studying malware samples, which embed malicious operations in components other that services. It is promising to identify how malware can hide their operations in other components or do long lasting operations, which mostly happen in malware. It is also important to extend our approaches by considering the temporal order of API calls. Currently we only include the APIs that invoked in the service components, without taking into account the call relationship among them. However, detecting repackaging by examining API call traces of the entire app (i.e., by considering all the app components) might turn to be challenging. Traces which contain sequence of API calls have historically been difficult to analyze due mainly to their size (as shown by Pirzadeh et al. 2010 and Pirzadeh et al. 2013). We need to push this research to determine (a) the complexity of API call traces of Android apps, and (b) the benefits of including all app component in the analysis.

Trace correlation: As discussed in Chapter 6, we use static analysis to generate API traces using the Soot tool. Such traces consists in an over-approximation of the set ordered sequences of API calls, up to a specified maximal length, that can be generated during a run of the target program. A more detailed understanding of the behavior of the underlying program could be gleaned by using traces that capture the complete sequence of API calls generated during a run of the target program. In our previous research (Hamou-Lhadj et al. 2013, Khoury et al. 2012), we showed how to detect the presence of malware by comparing multiple executions of different software that exhibit the same functionalities. However, it remains to be seen if this method could be adapted to the detection of adware, since the minimal changes exhibited between the benign and malicious versions of the app in the case could make detection difficult. Work on this topic is ongoing.

Repackaged apps in other platforms: In this thesis, we have limited our study to Android apps in part because there exists a large number of datasets of Android apps available to evaluate proposed approaches. Moreover, most of the studies in academic world are done on the Android platform. Note that Android is open-source, which makes it easier to study. Note also that malware use app repackaging in order to spread in other platforms such as IOS (OWASP 2016) as well. Therefore, it is promising to study the repackaged apps in other platforms. In those platforms, providing a dataset similar to AndroZoo, which contains pairs of benign apps and their repackaged version would greatly aid the research process

References

Aafer Y, Du W, Yin H (2013) DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android. In: International conference on security and privacy in communication systems. Springer, Cham, pp. 86–103

AdMob and AdSense policies. https://support.google.com/admob/answer/6128543?hl=en, Accessed 25 Feb 2019a

Alam MS, Vuong ST (2013) Random Forest Classification for Detecting Android Malware. In: 2013 IEEE international conference on green computing and communications and IEEE Internet of Things and IEEE cyber, physical and social computing, pp 663-669

Alazab M, Venkataraman S, Watters P (2010) Towards Understanding Malware Behaviour by the Extraction of API Calls. In: 2010 Second Cybercrime and Trustworthy Computing Workshop, IEEE, pp 52–59

Aldini A, Martinelli F, Saracino A, Sgandurra D (2015) Detection of repackaged mobile applications through a collaborative approach. Concurrency and Computation: Practice and Experience, Vol. 27, No. 11, pp 2818-2838

Android Developer Documentation (2018) https://developer.android.com/reference/dalvik/system/package-summary. Accessed 3 Mar 2018

Arora A, Garg S, Peddoju SK (2014) Malware Detection Using Network Traffic Analysis in Android Based Mobile Devices. In: 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies. IEEE, pp 66–71

Arp D, Spreitzenbarth M, Malte H, Gascon H, Rieck K (2014) DREBIN : Effective and Explainable Detection of Android Malware in Your Pocket. In: NDSS, Vol. 14, pp 23–26

Arzt S (2012) DroidBench, https://github.com/secure-software-engineering/DroidBench, Accesse 26 Jul 2018

Arzt S, Rasthofer S, Fritz C, Bodden E, Bartel A, Klein J, Le Traon Y, Octeau D, Mcdaniel P (2014) FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. ACM Sigplan Notes, Vol. 49, No. 6, pp 259–269

Au K W Y, Zhou YF, Huang Z, Lie D (2012) PScout : Analyzing the Android Permission Specification. In: CCS '12 Proceedings of the 2012 ACM Conference on Computer and Communications Security. pp 217–228

Ballano M (2011) Pjapps, https://www.symantec.com/connect/blogs/android-threats-getting-steamy, Accesse 26 Jul 2018

Bartel A, Klein J, Monperrus M, Traon Y Le (2012) Dexpler: Converting Android Dalvik Bytecode to Jimple for Static Analysis with Soot. In: ACM SIGPLAN International

Workshop on State of the Art in Java Program analysis. ACM, pp 27–38

Backes M, Bugiel S, Derr E (2016) Reliable Third-Party Library Detection in Android and its Security Applications. In: 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, pp. 356–367

Book T, Pridgen A, Wallach DS (2013) Longitudinal Analysis of Android Ad Library Permissions. IEEE Mobile Security Technology, ArXiv:1303.0857

Breiman L (2001) Random forests. Mach Learn, Vol. 45, No. 1, pp 5–32

Burguera I, Zurutuza U, Nadjm-Tehrani S (2011) Crowdroid: Behavior-Based Malware Detection System for Android. In: 1st ACM workshop on Security and privacy in smartphones and mobile devices - SPSM ’11. ACM, pp 15–26

Canfora G, Mercaldo F, Visaggio CA (2013) A classifier of Malicious Android Applications. In: Eighth International Conference on Availability, Reliability and Security (ARES). IEEE, pp 607–614

Chawla N V., Bowyer KW, Hall LO, Kegelmeyer WP (2002) SMOTE: Synthetic minority over-sampling technique. Journal of artificial intelligence research, 16, pp 321-357

Chen J, Alalfi MH, Dean TR, Zou Y (2015a) Detecting Android Malware Using Clone Detection. Journal of Computer Science and Technology, Vol. 30, No. 5, pp 942-956

Chen K, Liu P, Zhang Y (2014) Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In: 36th International Conference on Software Engineering - ICSE 2014. pp 175–186

Chen K, Wang P, Lee Y, Wang X, Zhang N, Huang H, Zou W, Liu P (2015b) Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale. In: 24th USENIX Security Symposium (USENIX Security 15). pp 659–674

Chen X, Li C, Wang D, Wen S, Zhang J, Nepal S, Xiang Y, Ren K (2019) Android HIV: A study of repackaging malware for evading machine-learning detection. IEEE Transactions on Information Forensics and Security, Vol. 15, pp 987-1001

Chien E (2005) Techniques of Adware and Spyware. In: Fifteenth Virus Bulletin Conference (Vol. 47). Dublin Ireland

Crussell J, Gibler C, Chen H (2015) AnDarwin: Scalable Detection of Android Application Clones Based on Semantics. IEEE Transaction on Mobile Computing, Vol. 14, No. 10, pp 2007–2019

Crussell J, Gibler C, Chen H (2012) Attack of the clones: Detecting cloned applications on Android markets. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp 37–54

Crussell J, Stevens R, Chen H (2014) MadFraud : Investigating Ad Fraud in Android Applications. In: 12th annual international conference on Mobile systems, applications, and services. ACM, pp 123–134

Desnos A (2015) Androguard: Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja !). https://github.com/androguard/androguard. Accessed 19 Jul 2018

dex2jar Tools (2018), https://github.com/pxb1988/dex2jar/wiki, Accessed 3 March 2018

Dominguez K (2011) BgServ, https://www.trendmicro.com/vinfo/us/threat-encyclopedia/malware/AndroidOS_BGSERV.A, Accessed 3 March 2018

Dong F, Wang H, Li L, Guo Y, Bissyande TF, Liu T, Xu G, Klein J (2018a) FraudDroid : Automated Ad Fraud Detection for Android Apps. In: 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp 257-268

Dong S, Li M, Diao W, Liu X, Liu J, Li Z, Xu F, Chen K, Wang X, Zhang K (2018b) Understanding Android Obfuscation Techniques : A Large-Scale Investigation in the Wild. In: International Conference on Security and Privacy in Communication Systems, Springer, Cham, pp 172-192

Enck W, Cox LP, Gilbert P, Mcdaniel P (2014) TaintDroid : An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. ACM Transaction on Computer Systems, Vol. 32, No.2, pp 1-29

Erturk E (2012) A Case Study in Open Source Software Security and Privacy : Android Adware. In: 2012 World Congress on Internet Security (WorldCIS). IEEE, pp 189–191

Fan W, Liu Y, Tang B (2015) An API calls monitoring-based method for effectively detecting malicious repackaged applications. International Journal of Security and Its Applications, Vol. 9, No. 8, pp 221-230

Fedler R, Kulicke M, Schütte J (2013) Native code execution control for attack mitigation on android. In: Third ACM workshop on Security and privacy in smartphones and mobile devices. ACM, pp 15-20

Fix E, Joseph L, Hodges J (1951) Discriminatory analysis-nonparametric discrimination: consistency properties. USAF school of Aviation Medicine

Forman I R, Forman N (2004) Java Reflection in Action. Manning Publications

Gao J, Li L, Tegawend PK (2019) Should You Consider Adware as Malware in Your Study ? In: 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp 604–608

Gascon H, Yamaguchi F, Rieck K, Arp D (2013) Structural Detection of Android Malware using Embedded Call Graphs Categories and Subject Descriptors. In: ACM workshop on Artificial intelligence and Security. ACM, pp 45–54

Gonzalez H, Kadir AA, Stakhanova N, Alzahrani AJ, Ghorbani AA (2014) Exploring Reverse Engineering Symptoms in Android apps. In: The Eighth European Workshop on System Security. ACM, pp 1-7

Google Inc. (2012) Cloud to Device Messaging (Deprecated).

https://developers.google.com/android/c2dm/. Accessed 23 Jul 2018

Grace M, Zhou Y, Zhang Q, Zou S, Jiang X (2012) RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In: 10th International Conference on Mobile Systems, Applications, and Services, pp 281–294

Guan Q, Huang H, Luo W, Zhu S (2016) Semantics-based repackaging detection for mobile apps. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pp 89–105

Gupta S. (2013) Types of Malware and its Analysis. In: International Journal of Scientific and Engineering Research, Vol. 4, No. 1

Hammad M, Garcia J, Malek S (2018) A Large-Scale Empirical Study on the Effects of Code Obfuscations on Android Apps and Anti-Malware Products. In: 40th International Conference on Software Engineering. ACM, pp 421–431.

Hamou-Lhadj A, Murtaza S. S, Fadel W, Mehrabian A, Couture M, Khoury R (2013) Software Behaviour Correlation in a Redundant and Diverse Environment Using the Concept of Trace Abstraction. In: Proc. of the ACM 2013 Research in Adaptive and Convergent Systems Conference (RACS'13), pp 328–335

Hanna S, Huang L, Wu E, Li S, Chen C, Song D (2013) Juxtapp: A scalable system for detecting code reuse among android applications. In: International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, Springer, Berlin, Heidelberg, pp 62-81

He Z, Ye G, Yuan L, Tang Z, Wang X, Ren J, Wang W, Yang J, Fang D, Wang Z (2019) Exploiting Binary-level Code Virtualization to Protect Android Applications Against App Repackaging. IEEE Access (Vo. 7), pp 115062-115074

Hu W, Tao J, Ma X, Zhou W, Zhao S, Han T (2014) MIGDroid: Detecting APP-Repackaging Android malware via method invocation graph. In: International Conference on Computer Communications and Networks, ICCCN. pp 1–7

Huang A (2008) Similarity Measures for Text Document Clustering. In: 6th new zealand computer science research student conference. pp 49–56.

Huang H, Zhu S, Liu P, Wu D (2013) A Framework for Evaluating Mobile App Repackaging Detection Algorithms. In: International Conference on Trust and Trustworthy Computing. Springer, Berlin, Heidelberg, pp 169–186

Hurier M, Suarez-Tangil G, Dash SK, Bissyande TF, Le Traon Y, Klein J, Cavallaro L (2017) Euphony: Harmonious Unification of Cacophonous Anti-Virus Vendor Labels for Android Malware. In: IEEE International Working Conference on Mining Software Repositories. pp 425–435

Islam R, Altas I (2012) A Comparative Study of Malware Family Classification. In International Conference on Information and Communications Security, Springer, Berlin, Heidelberg, pp 488-496

Jiang X (2011a) DroidKungFu, https://www.csc2.ncsu.edu/faculty/xjiang4/DroidKungFu.html, Accesse 26 Jul 2018

Jiang X (2011b) GingerMaster, https://www.csc2.ncsu.edu/faculty/xjiang4/GingerMaster/, Accesse 26 Jul 2018

Jiang X (2011c) HippoSMS, https://www.csc2.ncsu.edu/faculty/xjiang4/HippoSMS/, Accessed 26 Jul 2018

Jiang X (2011d) Plankton, https://www.csc2.ncsu.edu/faculty/xjiang4/Plankton/, Accessed 26 Jul 2018

Jiang X (2011e) SndApps, https://www.csc2.ncsu.edu/faculty/xjiang4/SndApps/, Accessed 26 Jul 2018

Jiao S, Cheng Y, Ying L, Su P, Feng D (2015) A Rapid and Scalable Method for Android Application Repackaging Detection. In: Lecture Notes in Computer Science. pp 349–364

Khanmohammadi K, Hamou-Lhadj A (2017) HyDroid: A Hybrid Approach for Generating API Call Traces from Obfuscated Android Applications for Mobile Security. In: IEEE International Conference on Software Quality, Reliability and Security (QRS), Prague, Czech Republic, pp 168-175

Khanmohammadi K, Ebrahimi N, Hamou-Lhadj A, Khoury R (2019a) Empirical Study of Android Repackaged Applications. Empirical Software Engineering, Vol. 24, No. 6, pp 3587-3629

Khanmohammadi K, Hamou-Lhadj A, Razgallah A, Khoury R (2019b) On the Use of API Calls to Detect Repackaged Malware Apps: Challenges and Ideas. In: the 30th International Symposium on Software Reliability Engineering (ISSRE 2019).

Khanmohammadi K, Rejali M, Hamou-Lhadj A (2015) Understanding the Service Life Cycle of Android Apps: An Exploratory Study. In: 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM), Denver, US, pp 81-86

Khoury R, Hamou-Lhadj A and Couture M (2012) Towards a Formal Framework for Evaluating the Effectiveness of Diversity when Applied to Security. In: Proc. of the IEEE Symposium on Computational Intelligence for Security and Defence Applications (CISDA'12), IEEE Computational Intelligence Society, pp 1-7

Kohavi R (1995) A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection. Ijcai, Vol. 14, No. 2, pp 1137–1145

Kornblum J (2006) Identifying almost identical files using context triggered piecewise hashing. In: Digital Investigation. pp 91–97

Khreich W, Khosravifar B, Hamou-Lhadj A, Talhi C (2017) An anomaly detection system based on variable N-gram features and one-class SVM. In: Elsevier Journal of Information & Software Technology (IST), 91: 186-197

Kumar M (2017) Beware! New Android Malware Infected 2 Million Google Play Store Users. https://thehackernews.com/2017/04/android-malware-playstore.html. Accessed 19 Jul 2018

Kywe SM, Li Y, Deng RH, Hong J (2014) Detecting camouflaged applications on mobile application markets. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp 241–254

Lee YK, Lim JD, Jeon YS, Kim JN (2014) Protection method from APP repackaging attack on mobile device with separated domain. In: International Conference on ICT Convergence. pp 667–668

Leka O (2016) Database of Android Apps | Kaggle. https://www.kaggle.com/orgesleka/android-apps/data. Accessed 19 Jul 2018.

Li L, Bissyandé TF, Klein J (2018) MoonlightBox: Mining Android API Histories for Uncovering Release-time Inconsistencies. In: 29th IEEE International Symposium on Software Reliability Engineering (ISSRE 2018), IEEE, pp 212-223

Li L, Bissyandé TF, Octeau D, Klein J (2016) DroidRA : Taming Reflection to Support Whole-Program Analysis of Android Apps. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, pp 318–329

Li L, Gao J, Hurier M, Kong P, Bissyandé TF, Bartel A, Klein J, Traon Y Le (2017a) AndroZoo++: Collecting Millions of Android Apps and Their Metadata for the Research Community. doi: 10.1145/2901739.2903508

Li L, Li D, Bissyande TF, Klein J, Le Traon Y, Lo D, Cavallaro L (2017b) Understanding Android App Piggybacking: A Systematic Study of Malicious Code Grafting. In: IEEE Transactions on Information Forensics and Security. IEEE, pp 359–361

Li L, Tegawendé Bissyandé, Jacques Klein(2019) Rebooting Research on Detecting Repackaged Android Apps: Literature Review and Benchmark, IEEE Transactions on Software Engineering

Li Y, Sundaramurthy SC, Bardas AG, et al. (2015) Experimental Study of Fuzzy Hashing in Malware Clustering Analysis. In: 8th Workshop on Cyber Security Experimentation and Test (CSET 15).

Lin YD, Lai YC, Chen CH, Tsai HC (2013) Identifying android malicious repackaged applications by thread-grained system call sequences. Computers and Security, Vol. 39, pp 340-350

Linares-Vásquez M, Holtzhauer A, Bernal-Cárdenas C, Poshyvanyk D (2014) Revisiting Android reuse studies in the context of code obfuscation and library usages. In: 11th Working Conference on Mining Software Repositories - MSR 2014. pp 242–251

Liu B, California S, Nath S, Nsdi I (2014) DECAF : Detecting and Characterizing Ad Fraud in Mobile Apps. In: 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14). pp 57–70

Luo L, Fu Y, Wu D, Zhu S, Liu P (2016) Repackage-proofing Android Apps. In: 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, pp 550–561

Mariconti E, Onwuzurike L, Andriotis P, De Cristofaro E, Ross G, Stringhini G (2017) MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models. In: 24th Network and Distributed System Security Symposium, arXiv preprint arXiv:1612.04433

Ma Z, Wang H, Guo Y, Chen X (2016) LibRadar : Fast and Accurate Detection of Third-party Libraries in Android Apps. In: the 38th international conference on software engineering companion. ACM, pp. 653–656

Maly F, Kriz P (2015) An Ad Hoc mobile cloud and its dynamic loading of modules into a mobile device running Google android. In: New Trends in Intelligent Information and Database Systems. Springer, Cham, pp 191–198

McAfee (2018) McAfee Threats Reports. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-dec-2018.pdf. Accessed 19 June 2019

Microsoft Advertising. https://advertising.microsoft.com/home. Accessed 25 Feb 2019b

Mojica IJ, Adams B, Nagappan M, Dienst S, Berger T, Hassan AE (2014) A Large Scale Empirical Study on Software Reuse in Mobile Apps. Software, IEEE Vol. 31, No. 2, pp 78–86

Mojica IJ, Nagappan M, Adams B, Hassan AE (2012) Understanding Reuse in the Android Market. In: 2012 IEEE 20th International Conference on Program Comprehension (ICPC), pp 113–122

Mulliner C, Robertson W, Kirda E (2014) VirtualSwindle : An Automated Attack Against In-App Billing on Android. In: 9th ACM symposium on Information, computer and communications security. ACM, pp 459–470

Nguyen T, Mcdonald J, Glisson W, Andel T (2020) Detecting Repackaged Android Applications Using Perceptual Hashing. In: 53rd Hawaii International Conference on System Sciences. In: Software Development for Mobile Devices, the Internet-of-Things, and Cyber-Physical Systems

Octeau D, Mcdaniel P, Bodden E (2013) Effective Inter-Component Communication Mapping in Android with Epicc : An Essential Step Towards Holistic Security Analysis. In: 22nd USENIX Security Symposium (USENIX Security 13). pp 543–558

OWASP (2016) Mobile Top 10 2016-Top 10 - OWASP. https://www.owasp.org/index.php/Mobile_Top_10_2016-Top_10. Accessed 19 Jul 2018

Pan X, Wang X, Duan Y, Wang X, Yin H (2017) Dark Hazard: Learning-based, Large-Scale Discovery of Hidden Sensitive Operations in Android Apps. In: 2017 Network Distribution System Security Symposium. doi: 10.14722/ndss.2017.23265

Petsas T, Voyatzis G, Athanasopoulos E, Polychronakis M, Ioannidis S (2014) Rage Against the Virtual Machine : Hindering Dynamic Analysis of Android Malware. In: the Seventh European Workshop on System Security. ACM, pp 1-6

Pirzadeh H, Shanian S, Hamou-Lhadj A, Alawneh A, Sharifee A (2013) Stratified Sampling of Execution Traces: Execution Phases Serving as Strata. In: Elsevier Journal of Science of Computer Programming, Special Issue on Software Evolution, Adaptability and Maintenance, 78(8), pp 1099–1118

Pirzadeh H, Agarwal A, Hamou-Lhadj A (2010) An Approach for Detecting Execution Phases of a System for the Purpose of Program Comprehension. In: the 8th International Conference on Software Engineering Research, Management & Applications (SERA 2010), pp 207 - 214

Poeplau S, Fratantonio Y, Bianchi A, Kruegel C, Vigna G (2014) Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In: NDSS, Vol. 14, pp 23–49

Potharaju R, Newell A, Nita-rotaru C, Zhang X (2012) Plagiarizing Smartphone Applications : Attack Strategies and Defense Techniques. In: International Symposium on Engineering Secure Software and Systems. Springer, Berlin, Heidelberg, pp 106–120.

Quinlan J. R (1986) Induction of Decision Trees. Machine Learning, Vol. 1, No. 1, pp. 81–106,

Ren C, Chen K, Liu P (2014) Droidmarking: Resilient SoftwareWatermarking for Impeding Android Application Repackaging. In Proc. of the 29th ACM/IEEE International Conference on Automated Software Engineering, pp 635–646

Rasthofer S, Arzt S, Miltenberger M, Bodden E (2015) Harvesting Runtime Data in Android Applications for Identifying Malware and Enhancing Code Analysis. Techincal Report, Technische Universität Darmstadt.

Rastogi V, Chen Y, Jiang X (2013) DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks. In: 8th ACM SIGSAC symposium on Information, computer and communications security, pp 329–334

Rastogi V, Shao R, Chen Y, et al (2016) Are these Ads Safe : Detecting Hidden Attacks through the Mobile App-Web Interfaces. In: The Network and Distributed System Security Symposium (NDSS)

Reina A, Fattori a, Cavallaro L (2013) A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. ACM European Workshop System Security (EuroSec), pp 1–6

Sahs J, Khan L (2012) A Machine Learning Approach to Android Malware Detection. In: European Intelligence and Security Informatics Conference. pp 141–147

Salem A, Banescu S, Pretschner A (2019) Don't Pick the Cherry: An Evaluation Methodology for Android Malware Detection Methods. arXiv preprint arXiv:1903.10560.

Sanz B, Santos I, Laorden C, Ugarte-Pedrero X, Bringas PG (2012) On the automatic categorisation of android applications. In: IEEE Consumer Communications and Networking Conference, CCNC 2012. pp 149–153

Saracino A, Sgandurra D, Dini G, Martinelli F (2016) MADAM : Effective and Efficient Behavior-based Android Malware Detection and Prevention. IEEE Transaction Dependable Secure Computing, Vol. 15, No. 1, pp 83–97

Schölkopf B, Smola AJ (2002) Support Vector Machines and Kernel Algorithms. Handb Brain Theory Neural Networks, pp 1119–1125

Symantec Report , (2010) Tapsnake, https://www.symantec.com/security_response/writeup.jsp?docid=2010-081214-2657-

99&tabid=2, Accesse 26 Jul 2018

Symantec Report , (2011a) GoldDream, https://www.symantec.com/security_response/writeup.jsp?docid=2011-070608-4139-99, Accesse 26 Jul 2018

Symantec Report , (2011b) NikySpy, https://www.symantec.com/security_response/writeup.jsp?docid=2011-072714-3613-99&tabid=2, Accessed 26 Jul 2018

Shahriar H, Clincy V (2014) Detection of repackaged Android Malware. In: 9th International Conference for Internet Technology and Secured Transactions.pp 349–354

Shannon C, Weaver W (1948) THE MATHEMATICAL THEORY OF COMMUNICATION. Bell System Technology journal, Vol. 27, No. 3, pp 379–423

Shao Y, Luo X, Qian C, Zhu P, Zhang L (2014) Towards a scalable resource-driven approach for detecting repackaged Android applications. In: 30th Annual Computer Security Applications Conference (ACSAC '14), pp 56–65

Sharif M, Lanzi A, Giffin J, Lee W (2008) Impeding Malware Analysis Using Conditional Code Obfuscation. In: Network and Distributed System Security Symposium (NDSS 2008)

Singhal A (2001) Modern Information Retrieval: A Brief Overview. IEEE Data Engineering Bull., Vol. 24, No. 4, pp 35-43

Soh C, Tan HBK, Arnatovich YL, Wang L (2015) Detecting Clones in Android Applications through Analyzing User Interfaces. In: 23rd IEEE International Conference on Program Comprehension. IEEE, pp 163–173

Sounthiraraj D, Sahs J, Garret G, Lin Z, Khan L (2014) SMV-HUNTER : Large Scale , Automated Detection of SSL / TLS Man-in-the-Middle Vulnerabilities in Android Apps. In: 21st Annual Network and Distributed System Security Symposium (NDSS'14)

Statista (2018) Rating of apps on Google Play as of May 2018. https://www.statista.com/statistics/266217/customer-ratings-of-android-applications. Accessed 26 Jul 2018

Suarez-Tangil G, Tapiador JE, Peris-Lopez P, Blasco J (2014) Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. Expert Systems with Applications, Vol. 41, No. 4, pp.1104–1117, doi: 10.1016/j.eswa.2013.07.106

Sun M, Li M, Lui JCS (2015) DroidEagle: seamless detection of visually similar Android apps. In: 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks. ACM, pp 1-12

Sun M, Li X, Lui JCS, Ma RTB, Liang Z (2017) Monet: A User-oriented Behavior-based Malware Variants Detection System for Android. IEEE Transaction on Information Forensics and Security, Vol. 12, No. 5, pp 103–1112.

Sun X, Zhongyang Y, Xin Z, Mao B, Xie L (2014) Detecting Code Reuse in Android Applications Using Component-Based Control Flow Graph. In: International Information Security and Privacy Conference. pp 142–155

Statiscas (2019) Statistics and Market Data on Mobile Internet & Apps, http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/. Accessed 4 July 2019

Symantec Report (2014) Android.Appenda. https://www.symantec.com/security-center/writeup/2012-062812-0516-99. Accessed 4 Mar 2019

Takabi H, Joshi JBD, Ahn GJ (2010) SecureCloud: Towards a comprehensive security framework for cloud computing environments. Proc - Int Comput Softw Appl Conf :393–398 . doi: 10.1109/COMPSACW.2010.74

Tam K, Khan SJ, Fattori A, Cavallaro L (2015) CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In: 2015 Network and Distributed System Security Symposium

Tanner S, Vogels I, Wattenhofer R (2019) Protecting Android Apps from Repackaging Using Native Code. In: 12th International Symposium on Foundations & Practice of Security (FPS 2019).

Tian K, Yao D, Ryder BG, Tan G (2016) Analysis of Code Heterogeneity for High-Precision Classification of Repackaged Malware. In: IEEE Symposium on Security and Privacy Workshops, SPW 2016. pp 262–271

Viennot N, Garcia E, Nieh J (2014) A measurement study of google play. In: 2014 ACM international conference on Measurement and modeling of computer systems, pp. 221-233, doi: 10.1145/2591971.2592003

VirusTotal (2018) Free Online Virus Malware and URL Scanner. In: Google Inc. https://www.virustotal.com/#/home/upload. Accessed 19 Jul 2018

Wang H, Guo Y, Ma Z, Chen X (2015) WuKong: a scalable and accurate two-phase approach to Android app clone detection. In: International Symposium on Software Testing and Analysis (ISSTA 2015), pp. 71–82

Winter C, Schneider M, Yannikos Y (2013) F2S2: Fast forensic similarity search through indexing piecewise hash signatures. Digital Investig, Vol. 10, No. 4, pp. 361–371, doi: 10.1016/j.diin.2013.08.003

Wiśniewski R (2012) Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps. https://ibotpeaches.github.io/Apktool/. Accessed 19 Jul 2018

Wong MY, Lie D (2016) IntelliDroid : A Targeted Input Generator for the Dynamic Analysis of Android Malware. In: NDSS, Vol. 16, pp. 21-24

Wu DJ, Mao CH, Wei TE, Lee HM, Wu KP (2012) DroidMat: Android malware detection through manifest and API calls tracing. In: 2012 7th Asia Joint Conference on Information Security, AsiaJCIS 2012. pp 62–69

Wu X, Zhang D, Su X, Li W (2015) Detect repackaged Android application based on HTTP traffic similarity. Security and Communication Networks, Vol. 8, No. 13, pp. 2257–2266, doi: 10.1002/sec.1170

Xia M (2015) BeanBot, https://github.com/mingyuan-xia/AppAudit/wiki/BeanBot-analysis-report, Accesse 26 Jul 2018

Xia M, Gong L, Lyu Y, Qi Z, Liu X (2015) Effective Real-time Android Application Auditing. In: 2015 IEEE Symposium on Security and Privacy. IEEE, pp 899–914

Xue Y, Meng G, Liu Y, Tan TH, Chen H, Sun J, Zhang J (2017) Auditing Anti-Malware Tools by Evolving Android Malware and Dynamic Loading Technique. IEEE Trans Inf Forensics Secur 12(7):1529–1544 . doi: 10.1109/TIFS.2017.2661723

Xu K, Li Y, Deng RH (2016) ICCDetector: ICC-Based Malware Detection on Android. IEEE Transaction on Information Forensics and Security, Vol. 11, No. 6, pp. 1252–1264, doi: 10.1109/TIFS.2016.2523912

Yang C, Xu Z, Gu G, Yegneswaran V, Porras P (2014) DroidMiner : Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android. In: European Symposium on Research in Computer Security. pp. 163–182

Yang Z, Yang M, Wang XS (2013) AppIntent : Analyzing Sensitive Data Transmission in Android for Privacy Leakage Detection. In: 2013 ACM SIGSAC conference on Computer and communications security. ACM, pp. 1043–1054

Yerima SY, Sezer S, McWilliams G, Muttik I (2013) A New Android Malware Detection Approach Using Bayesian Classification. In: 27th International Conference on Advanced Information Networking and Applications (AINA). IEEE, pp. 121–128

Yoshikawa T (2012) GPSSMSpy, http://blog.trendmicro.com/trendlabs-security-intelligence/beta-version-of-spytool-app-for-android-steals-sms-messages/, Accessed 26 Jul 2018

Yue S, Feng W, Jiang Y, Tao X, Xu C, Lu J (2017) RepDroid: an automated tool for Android application repackaging detection. In: IEEE/ACM 25th International Conference on Program Comprehension (ICPC 2017). IEEE, pp. 132–142

Zeng Q, Luo L, Qian Z, Du X, Li Z (2018) Resilient Decentralized Android Application Repackaging Detection Using Logic Bombs. In: 2018 International Symposium on Code Generation and Optimization, ACM, pp 50–61

Zeng Q, Luo L, Qian Z, Du X, Li Z, Huang C T, Farkas C (2019) Resilient User-Side Android Application Repackaging and Tampering Detection Using Cryptographically Obfuscated Logic Bombs. IEEE Transactions on Dependable and Secure Computing.

Zhang F, Huang H, Zhu S, Wu D, Liu P (2014) ViewDroid: Towards obfuscation-resilient mobile application repackaging detection. WiSec 2014 - Proc 7th ACM Conf Secur Priv Wirel Mob Networks :25–36 . doi: 10.1145/2627393.2627395

Zhang L, Niu Y, Wu X, Wang Z, Xue Y, Science C, College T (2013) A3 : Automatic Analysis of Android Malware. In: 1st International Workshop on Cloud Computing and Information Security, Atlantis Press, pp. 89–93

Zhauniarovich Y, Gadyatskaya O, Crispo B, La Spina F, Moser E (2014) FSquaDRA: Fast detection of repackaged applications. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). pp. 130–145

Zhao Y, Qian Q (2018) Android Malware Identification Through Visual Exploration of Disassembly Files. Journal Network Security, Vol. 20, No. 6, pp. 1005–1015, doi: 10.6633/IJNS.201811

Zheng C, Zhu S, Dai S, Gu G, Gong X (2012) SmartDroid : an Automatic System for Revealing UI-based. In: the second ACM workshop on Security and privacy in smartphones and mobile devices, ACM, pp 93–104

Zhou W, Wang Z, Zhou Y, Jiang X (2014) DIVILAR: Diversifying Intermediate Language for Anti-repackaging on Android Platform. In: 4th ACM conference on Data and Application Security and Privac (CODASPY '14). pp 199–210

Zhou W, Zhang X, Jiang X (2013a) AppInk : Watermarking Android Apps for Repackaging Deterrence. In: 8th ACM SIGSAC symposium on Information, Computer and Communications Security. pp 1–12

Zhou W, Zhou Y, Grace M, Jiang X, Zou S (2013b) Fast , Scalable Detection of " Piggybacked " Mobile Applications. In: the third ACM conference on Data and application security and privacy, ACM, pp 185–196

Zhou W, Zhou Y, Jiang X, Ning P (2012a) Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In: the second ACM conference on Data and Application Security and Privacy, ACM, pp 317–326

Zhou Y, Jiang X (2012a) Android Malware Genome Project. http://www.malgenomeproject.org/ Accessed 10 Jul 2015

Zhou Y, Jiang X (2012b) Dissecting Android Malware: Characterization and Evolution. In: 2012 IEEE Symposium on Security and Privacy. IEEE, pp 95–109

Zhou Y, Jiang X (2013) Detecting Passive Content Leaks and Pollution in Android Applications. In: 20th Network and Distributed System Security Symposium (NDSS).

Zhou Y, Wang Z, Zhou W, Jiang X (2012b) Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. 19th Annual Network Distribution Systems Security Symposium (NDSS 2012), Vol. 25, No. 4, pp 50–52