8-2016

pDroid

Joe Larry Allen
University of Tennessee, Knoxville, jallen89@vols.utk.edu

To the Graduate Council:

I am submitting herewith a thesis written by Joe Larry Allen entitled "pDroid." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Engineering.

Jinyuan Sun, Major Professor

We have read this thesis and recommend its acceptance:

Qing Cao, Michael Jantz

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

pDroid

A Thesis Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Joe Larry Allen

August 2016

Abstract

When an end user attempts to download an app on the Google Play Store they receive two related items that can be used to assess the potential threats of an application, the list of permissions used by the application and the textual description of the application. However, this raises several concerns. First, applications tend to use more permissions than they need and end users are not tech-savvy enough to fully understand the security risks. Therefore, it is challenging to assess the threats of an application fully by only seeing the permissions. On the other hand, most textual descriptions do not clearly define why they need a particular permission. These two issues conjoined make it difficult for end users to accurately assess the security threats of an application. This has lead to a demand for a framework that can accurately determine if a textual description adequately describes the actual behavior of an application. In this Master Thesis, we present pDroid (short for privateDroid), a market-independent framework that can compare an Android application's textual description to its internal behavior. We evaluated pDroid using 1562 benign apps and 243 malware samples, and pDroid correctly classified 91.4% of malware with a false positive rate of 4.9%.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Chapter 1 discusses the need for a framework that can compare an application's alleged behavior to its actual behavior (Section 1.1). Section 1.2 discusses the contributions made in this master thesis. Finally, this chapter concludes by describing the remaining structure of this master thesis.

1.1 Motivation

A growing concern among smartphone users is how third-party applications handle sensitive data, and a recent survey on smartphone users found that 19 out of the top 25 user concerns were related to leaking or tampering with user information [19]. When an end user installs an application from the Google Play Store, they receive metadata about the application, including the textual description provided by the developer and the list of permissions used by the application. To protect their information, they must use the textual descriptions and list of permissions to assess any security threats properly. However, it has been shown that end users lack the ability to comprehend the security risks related to Android permissions [22] and only 9.1% of application descriptions appropriately describe the need for all the permissions used in the application [52]. These two issues conjoined make it difficult for end users to assess the security threats of an application, and it has created a desire

for a framework that can evaluate if an application description adequately describes the applications actual behavior.

To address this concern, researchers have developed several frameworks to compare an app's textual descriptions to its actual behavior [29, 41, 46, 52, 64]. These papers mainly have the same goal, but how they describe actual behavior varies. The work in [46, 52, 64] use the Android permissions an application uses to describe the actual behavior. The techniques utilized in these papers attempt to evaluate the textual description and verify if the description states a need for the permissions requested by the app. One of the concerns with this method is that defining actual behavior by permissions is a too coarse-grained approach. To create a finer-grained approach the work in [29, 41] extracted the sensitive APIs used in an application to define its actual behavior. However, just because an application uses a dangerous API doesnt necessarily mean it has malicious intent. The framework proposed in this master thesis, pDroid, attempts to capture an even finer-grained image of an application's behavior pDroid uses the dataflows within an application to define the actual behavior of an application. APIs can only tell us an application accessed sensitive information. However, dataflows can tell us what sensitive information was accessed and how an application handled this information. For example, while [29, 41] may consider an app to be suspicious because it uses a sensitive API, such as *LocationManager.getLastLocation()*. pDroid will further investigate by checking what sink received the information. pDroid would most likely consider the behavior of writing the user's location to a log file as not suspicious while, on the other hand, writing the location to a buffer and sending it to a private server would be suspicious.

1.2 Contributions

We created pDroid (short for privateDroid), a market-independent framework that can compare an Android application's textual description to its actual internal behavior. Unlike previous approaches that use permissions or APIs [29, 41, 46, 52,

64], pDroid defines an application's actual behavior by the dataflows extracted using static-taint analysis. We evaluated pDroid using 1562 benign apps and 243 malware samples, and pDroid correctly classified 91.4% of malware, with a false positive rate of 4.9%.

1.3 Structure of Thesis

The remaining chapters cover the background, technical details, and evaluation of pDroid. Chapter 2 covers several background topics related to pDroid and why we chose to focus on the Android platform. In Chapter 3 we discuss the technical details related to clustering applications based on their textual descriptions. Chapter 4 discusses how we identify anomalies based on inconsistencies between the textual description and the dataflows within an application. Our evaluation results and how we developed our dataset can be found in Chapter 5. Finally, this masters thesis concludes in Chapter 6.

Chapter 2

Background

2.1 Android

The Android operating system (OS) was unveiled in 2007 and is maintained by the Open Handset Alliance under the direction of Google. Android is based on the Linux kernel and is designed primarily for touchscreen devices such as smartphones and tablets. Currently, Android is the most used mobile OS worldwide and held 80.7% of the fourth quarter 2015 market shares for mobile operating systems [28]. The Android OS allows end users to install applications onto their mobile device that extends the functionality of the device. The most common method for end users to find applications that meet their needs is to search an application marketplace. The largest application market is GooglePlay* with over two million apps as of 2016 [4]. A centralized approach to distributing applications is efficient for both developers and end users. It allows developers a fast, efficient, and simple method to reach millions of potential customers. For an end user, a centralized approach allows them to search quickly for applications that can meet their personal needs. While this approach is useful from a distrubution and marketing perspective, it can be overwhelming for the operator's of these marketplaces to properly vet the actual

*https://play.google.com

intent of applications uploaded onto the marketplace. Therefore, centralized markets have created an environment for malware developers to exploit with many potential victims.

2.1.1 Android Architecture

The Android Architecture is made up of several layers including the application layer, Android framework, native libraries and Android runtime, hardware abstraction layer, and the Linux kernel. The layers make up the Android software stack, which is shown in Figure 2.1. The lowest layer is the *Linux kernel* and is the foundation of the Android platform. The Linux kernel handles low-level system services, such as memory and process management, networking, and controlling hardware drivers. The kernel also provides Android with several security features such as user-based permission models and process isolation. The *hardware abstraction layer (HAL)* defines a standard interface for hardware vendors to implement and allow Android to be agnostic about lower-level driver-implementations. The next layer includes the Android runtime environment which encompasses the Dalvik Virtual Machine (DVM) and core libraries. Programs for Android are commonly written in Java and compiled to bytecode for a Java Virtual Machine (JVM), which is then translated into Dalvik bytecode and stored in a Dalvik Executable (.dex) file allowing for execution on the DVM. The Dalvik Executable was optimized for minimal memory footprint allowing it to be ideal for an OS targeted towards devices with constrained processing power, memory, and storage [17]. Each application running on an Android device is isolated within its own DVM. The core libraries fall into three categories: Dalvik VM-specific libraries, Java interoperability libraries, and Android libraries. The Dalvik VM specific libraries are the set of libraries used predominantly for interacting with the Dalvik VM; the Java interoperability libraries are an open source implementation of a subset of the standard Java core libraries that have been adapted for use by applications running in a DVM. The Android libraries are the Java-based libraries

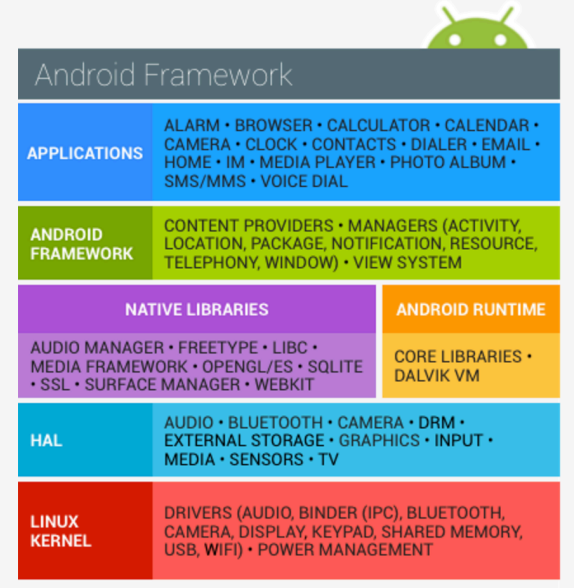


Figure 2.1: The Android Stack Software Stack.

*Android. *Android Security*. Accessed: 5-24-2016. URL: https://source.android.com/security/

that are for Android development (networking, databases, graphics, os, etc.). The final discussion about this level is the *native libraries*, which are written in C/C++. The Android libraries discussed previously are Java-based. However, many of the APIs called from the Android libraries do not do the majority of the work but are Java-based wrappers around a set of native C/C++ libraries. These C/C++ libraries are included to fulfill a extensive and diverse range of functions such as SQLite database management, audio, and Secure Socket Layer (SSL) communication [58]. The next level of the software stack is the *Android framework*, which provides interfaces that form the environment Android applications are running upon; this includes Android API's that manage location, activities and content providers. On top of the software stack are the *Android applications*. Including all applications, such as home, browse, and third-party applications.

2.1.2 Android Applications

Android applications are written using the Java programming language. Unlike conventional Java programs, Android applications do not have a *main()* function or a single entry point for execution. Instead, they are designed using components. App components make up the essential building blocks of an Android app. Each component is a different point through which the system can enter a developer's application. There are four different types of components: activities, services, content providers, and broadcast receivers. Each type of component serves a different role and the set of components used in an Android application define its overall behavior. The *activity* component creates user interfaces. For example, a messaging application may have one activity that creates the user interface for allowing a user to input their message and another activity for allowing the user to view their contacts. The *service* component runs in the background to perform tasks. Unlike, activity components, service components do not have a user interface. For example, a service component can be used to play music in the background. The *content*

provider component handles application data. Using content providers, an application can store data in files, SQLite databases, or other persistent storage locations an application can access. The *broadcast receiver* component responds to system-wide broadcast announcements. For example, the system may broadcast that a picture has been captured, and the broadcast receiver can alert the application of this action. In general, broadcast receivers do minimal work, but instead, alert other components that an event occurred.

2.1.3 Android Security

The Android mobile platform was designed to be truly open. Android applications are given access to hardware and software, local data and data from servers. This type of open platform requires a strong security architecture, and Android was developed with multi-layered security that allows for an open environment while protecting the users of the platform [1]. The majority of security between Android applications and the Android system is enforced at the process level using standard Linux facilities, such as assigning unique user IDs to each Android application. The *Linux kernel* has been in widespread use for years and used in millions of security-sensitive environments. It also has consistently been researched, attacked, and fixed by thousands of developers allowing it to become a stable and secure kernel. The Linux kernel provides Android with several security features such as a user-based permission model, process isolation, and the ability to remove insecure parts of the kernel. The Linux OS is a multiuser OS and enforces the security policy that a user's resources and data must be protected from other users using the system. Android leverages Linux's user-based protection scheme by implementing the policy that application resources must be protected from other applications running on the system. To achieve this at the kernel level, each application is running as a separate process with its own unique user ID (UID). Therefore, the code of two different applications cannot run in the same address space. Access-control is provided through a permission mechanism that enforces

restrictions on the operations a particular application can perform. By default, an Android application has no permissions associated with it, meaning it cannot do anything adversely impact the user experience or any data on the device [1]. If an app developer wishes to have access to a user's sensitive data, they must declare they are using the permission in the app's manifest file. For example, an application that needs to monitor incoming SMS messages would need to request access for the Android.permission.RECEIVE_SMS permission.

2.2 Mobile Malware

Cabir [60] was the first known malware written specifically for smartphones and was discovered by a cyber security and privacy company, F-Secure,[†] in June 2004 [32]. Cabir was a proof-of-concept worm that used Bluetooth to propagate itself to new victims. A novel feature of Cabir was that it did not exploit vulnerabilities found in the system, and it worked within the security parameters of the Symbian OS. Instead, it manipulated the user interface to force users of infected phones to propagate the worm to victims in range. When a potential victim was in Bluetooth range, the infected phone would incessantly display file-transfer requests on the infected device's screen requiring the user to approve or deny the request. If the user chose to deny the request, Cabir would display another file-transfer request, and once again require the user to approve or deny the request. To end the constant file-transfer requests, the user would eventually need to accept the file-transfer, allowing Cabir to infect the new victim.

While Cabir and modern malware both aspire to manipulate end users into executing their malicious payload, the mobile ecosystem Cabir exploited pales in comparison to the current mobile ecosystem. First, the number of users has increased significantly. In 2005, just 2% of the U.S. population used a smartphone [14], while in 2016 66% of the U.S. population uses smartphones [43]. The growth of mobile malware

[†] https://www.f-secure.com/en_US/welcome

has also substantially increased. In June of 2005, there were only approximately 60 known mobile malware programs [32]. In 2015, Kaspersky Lab's detected 2,961,727 malicious mobile packages and 884,774 new malicious mobile programs [61]. Finally, the modern smartphone have far more computational power and access to private data compared to the smartphone in 2004.

The modern phone ecosystem contains a variety of mobile operating systems, but the two primary operating systems are Google's Android and Apple's iOS. Both operating systems allow end users to install third-party applications, but the procedures used by Apple and Google to regulate untrusted third-party applications are fundamentally different. For iOS, a third-party application must pass a stringent human-guided review process, before it is approved to run on iOS devices. Third-party developers are required to distribute their application through the official iOS application marketplace, App Store. If a third-party developer wishes to distribute his app in the App Store, he must first submit it to Apple for human evaluation, and the application is only published to the App Store after it has passed the review process. The technical details of the review process are largely unknown, but according to the official Apple App Review Guidelines [3], developers should expect their apps to go through a thorough inspection for all possible term violations. If an application is approved, it will be signed and published to the App Store. Additionally, Apple enforces a mandatory code signing mechanism to ensure only applications that have been approved and signed by Apple are allowed to run on iOS devices, preventing end users from installing unapproved applications. The implementation details of the code signing mechanism can be found in [44]. In constrast, the Android mobile platform was developed to be truly open, and Google takes a more permissive stance on regulating untrusted third-party applications. The offical Android app marketplace is Google Play. However, unlike iOS, Android third-party developers are not required to distribute their application via Google Play and do not need Google's approval before distributing their application. This has created several "alternative" and "unofficial" app marketplaces, and third-party developers can choose to distribute

their applications through these markets instead. Third-party app stores are growing in popularity and provide great convenience to users, especially in countries where official stores are not available. For example, the largest third-party marketplace is 9Apps[‡], which has 250 million monthly users as of April 2016 and over 26 million daily app downloads [12]. Finally, Android users have the freedom to download apps from any source they choose, unlike iOS users who are limited to only downloading Apple approved applications from the App Store.

While iOS has been criticized for developing an approval process that is opaque, arbitrary, and limits freedom of expression [31], the review process has to prevent malicious applications from being published in the App Store. Despite iOS's popularity, only a handful of malicious apps for iOS have been discovered [20], but like all security methods, the Apple's review process is not completely secure. Wang et. al. successfully published the proof-of-concept malware *Jekyll* [62] into the App Store. The malicious app was successfully able to perform many malicious tasks, such as stealthily post tweets, take photos, send email, and exploit kernel vulnerabilities. The fundamental idea behind Jekyll was to make it remotely exploitable and introduce malicious control flows by rearranging signed code. Since the malicious control flows are added after Apple's approval and did not exist during the review process, Jekyll stayed undetected.

The vision to make Android truly open by the Open Handset Alliance and Google has allowed the Android OS to be the most used OS worldwide [28]. Unfortunately, its lack of regulation has several security drawbacks and has made Android the ideal target for malware developers. In 2011, Google addressed the problem of Android malware by introducing Bouncer [38], an automatic dynamic analysis tool used to scan new and existing Android applications. Unfortunately, the work of Percoco et. al. [48] found that Bouncer could be easily by-passed by discovering the emulation environment, such as the IP Address of the emulator, and preventing the malicious payload from executing during an inspection. In 2015 Google announced they had

[‡]http://www.9apps.com/

added an additional layer of security by having an internal team analyze apps for policy violations before being published on the Google Play Store, similar to the approach Apple uses [49]. However, Google's new layers of security only regulates the Google Play Store and is unlikely to thwart the trend of Android malware. This is because the overwhelming majority of Android malware is being developed and distributed in unregulated third party App Stores in the Middle East and Asia [51], such as 9Apps. To better understand the security status of Android App Stores, Cheeta Mobile Security[§] surveyed several top alternative app stores, and found that malware is found at a much higher rate, shown in Table 2.1. Out of the sample of apps chosen from Google Play, only 48 samples (0.005%). In contrast, out of the samples from 9Apps, 0.16% apps contained malware.

Unlike the iOS platform, Android user's have the freedom to download a third-party application from unoffical markets. Third-party app stores bring great convenience to users, especially in countries where official App Stores are unavailable. However, the lack of regulations has made the Android platform the ideal target for malware developers, and the overwhelming majority of mobile malware is found on Android. Therefore, the remaining discussion in this master thesis will be focused solely on the Android platform and Android malware.

2.3 Related Work

The components of pDroid touches several research areas such as Android security, natural language processing (NLP) in software engineering, and app store mining. To our knowledge, the first proposed framework for comparing the textual description to an application's behavior was Whyper [46]. Whyper correlates the description and permission by extracting natural language keywords from Android API documents. Since APIs and permissions can be related together [6], the intuition is that patterns expressed in the API documentation will have a presence in the application

[§] http://www.cmcm.com/en-us/

Table 2.1: Number of samples and malwares in third-party App Stores

App market	Number of Sample	Number of malware	Percentage of malware
9apps	32698	53	0.16%
Getjar	1865	3	0.16%
Vshare	14196	13	0.09%
Aptoide	37098	20	0.05%
Mobogenie	23001	9	0.04%
Google Play	904464	48	0.005%

*Cheeta Mobile. *Android App Stores Become Significant Source for Malware*. Accessed : 6-8-2016. 2016. URL: http://www.cmcm.com/blog/en/security/2016-01-20/925.html

description, which implies the need for a given permission. If the presence exists, the application description is transparent and appropriately describes the application's behavior; if not, it should raise suspicion.

Three major limitations of Whyper are *limited semantic information*, *lack of associated APIS*, and *lack of automation*. The limitation of semantic information is because Whyper uses API documentation to extract semantic patterns. Since API documentations are related to only describing an API's functionality, it cannot obtain subtle correlations between the textual descriptions and the declared permissions. For example, the work of Qu et. al [52] found that words related to banking, such as "deposit" and "check" are related to the CAMERA permission, because banking applications allow end users to deposit checks by snapping a picture. This type of inference cannot be inferred using only API documentations. Next, certain sensitive permissions do not have any related APIs [6], such as RECEIVE_BOOT_COMPLETED. Therefore, it is not possible to extract an semantic information related from them. Finally, Whyper's technique is not fully automated and requires manual extraction of patterns from API documents.

The work of Qu et. al. addressed these limitations by proposing Autocog [52], a fully automated technique for comparing application descriptions to declared permissions. To prevent *semantic limitation* and *lack of associated APIs*, Autocog extracts semantic information from textual descriptions instead of API documentation. To provide automation, Autocog leveraged Explicit Semantic Analysis [27] to extract semantic information from textual descriptions. Autocog was a significant improvement over Whyper, but it also has its limitations. First, AutoCog only supports 11 permissions and does not handle critical permissions that are related to privacy leaks (e.g., phone number, device identifier, service provider, etc.), sending and receiving text messages, network I/O and critical system-level behaviors. Additionally, the work of Zhang et. al. discovered that Autocog sometimes cannot recognize certain words that have substantial security implications. For example, if "geographic location" is used to describe the permissions

ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION AutoCog cannot associate this phrase with any permissions.

Both Whyper and Autocog use permissions to define actual behavior. However, using permissions to define the behavior of an application is a coarse grained approach that raises several concerns. First, it is often the case that Android applications request more permissions than they actually use [6, 21, 59]. Next, it has been shown that end users lack the ability to comprehend the security threats related to Android permissions [21]. Finally, Qu et. al. found that only 9.1% of textual descriptions properly describe the need for all of their permissions [52].

Most similar to pDroid is Chabada [29], which compares the textual descriptions to the actual behavior. Unlike Whyper and AutoCog, Chabada defines actual behavior as the sensitive APIs found within an application. Using the APIs, Chabada can provide a clear picture of the application's actual behavior. The intuition behind Chabada is that applications with similar textual descriptions should have similar API usage. Like pDroid, Chabada clusters applications based on their textual descriptions. Then it searches for anomalous API usage in application clusters. We see pDroid as a natural extension to Chabada that provides a more in-depth description of the application's actual behavior. The biggest difference between Chabada and pDroid is that pDroid leverages a more fine-grained approach (dataflows) to define an application's behavior. The advantage of this method is that reports of anomalous behavior can provide a clearer picture of the malicious intent. Additionally, pDroid leverages binary classification, which labels an application as benign or malicious, while Chabada uses a one-class SVM for detecting anomalies. In general, binary classification provides higher accuracy on known malware, while anomaly detection is better suited for potential zero-day attacks.

In Addition to Chabada, pDroid has similar goals as AAPL [39]. AAPL uses conditional data flow analysis and joint data flow analysis to find data leakages in apps. AAPL employees peer voting to distinguish legitimate privacy disclosures from malicious data leaks. The idea behind peer voting is that applications with

similar functionality should exhibit similar privacy disclosures. To obtain peers, AAPL leverages Google Play's recommendation system to get peer applications. In their work, they found Google Play's recommendation system provided more similar applications than textual descriptions alone. However, this type of clustering is not suitable for pDroid. First, we envision pDroid assisting end users in assessing the security threats related to the permissions and textual descriptions. Second, pDroid was designed to be market independent, and using the Google Play's recommendation system would violate this policy.

Chapter 3

Clustering Applications Using Alleged Behavior

The intuition behind pDroid is that applications with similar alleged behavior (textual descriptions) should have similar actual behavior (dataflows). In this chapter, we discuss the technical details and algorithms used to cluster applications with similar textual descriptions. Section 3.1 presents the detailed preprocessing phase all textual descriptions go through before being clustered. In Section 3.2 we discuss how Latent-Dirichlet Allocation (LDA) [11] is used to convert textual descriptions into document-topic distributions. Finally, Section 3.3 we discuss how pDroid leverages the document-topic distributions to cluster applications by the semantic similarity between their textual descriptions using Affinity Propagation [23].

3.1 Preprocessing Application Descriptions

When a developer wants to publish his application, he will upload his application to the Google Play Store. Along with the application, the developer will also upload metadata. This metadata includes a textual description, images, the application name, etc. In pDroid, we are concerned with the textual description of an

application, and we consider this the *alleged behavior* of an application. To allow for personalization and customization, the Google Play Store has taken a liberal stance regarding format for application descriptions. While this lenient format is ideal for creativity, it can cause issues for natural language processing (NLP) tools. Therefore pDroid first sends all application descriptions through a stringent preprocessing phase before analysis is completed.

When an unseen app is first introduced to pDroid, its textual description will go through a sanitizing process. The Google Play Store provides support for many languages, and the current version of pDroid only provides support for English. Therefore, pDroid removes any nonEnglish descriptions using Chromium Language Detector [15], which can detect over 80 different languages. After this step is completed, we use Python's powerful Natural Language Toolkit (NLTK) [10] to do the remaining of our NLP preprocessing. In our first step, we reduce the number of distinct words in the dataset using case folding, so "The", "tHE", and "THE" all become "the". Many developers place URLs within their application description to provide users with the location of external information regarding their application, company, or personal web page. These URLs are unique and provide little information about the *alleged behavior* of an application. Therefore we remove all URLs in a textual description. Next, we remove any punctuation characters, non-alphabetic characters, and any non-ASCII values such as ♣, ♠, and ©. We then remove any word that is less than three characters, so any possible artifacts from the previous sanitizing processes are eliminated. Topic Models are useful for extracting patterns for meaningful word use, but they are not good at determining which words are meaningful. It is often the case that the use of very common words such as "the", "you", and "have" do not indicate the type of similarity between documents that we are interested in [53]. Therefore, we remove all common English stopwords. Words that are ubiquitous in our dataset such as "app","Google", and"Android" tell us little about the similarity between application descriptions, therefore pDroid removes the 20 most common words found in the training dataset. Words found in application

descriptions that are exceedingly rare, such as the application developer's name, also provide little information related to the similarity of application descriptions and we remove any word that does not occur in more than two application descriptions. Terms such as"charge", "charged", and "charging" are similar, and we want them to be represented by the same word, and pDroid uses a Porter Stemmer [50] for term normalization.

Any application description that is less than ten words after the preprocessing phase is removed from the dataset. These application descriptions do not provide enough context to allow pDroid to infer their alleged behavior. If pDroid is being used by an application market management team, it is advised that the management team requests the developer of the application to provide an application description of better quality before they approve the application to their marketplace. Figure 3.1 shows the Pinterest Android application* description before sanitization and Figure 3.2 shows the application description after sanitization.

3.2 Converting Textual Descriptions to Feature Vectors Using LDA

In Google Play, the developer chooses a category ("Social", "Music & Audio", "Tools", etc.) to assign their application. The simplest approach for clustering applications would be to cluster applications based on the category they were assigned to in the Google Play Store. However, using this method has several shortcomings. First, previous work that has clustered applications based on textual descriptions has found that the categories inferred from their analysis did not match the categories found in the Google Play Store. For example, the application categories found in the work of Gorla et. al. found that applications that do nothing but display ads and typically promise the user with some benefit form the "advertisement" category

*https://play.google.com/store/apps/details?id=com.pinterest

Pinterest is a visual bookmarking tool that helps you discover and save creative ideas. Use Pinterest to make meals, plan travel, do home improvement projects and more. With Pinterest you can:

- Plan a project: Home remodels, garden redesigns and other DIYs
- Get creative ideas: Recipes to cook, articles to read, gifts to buy and ways to save money
- Explore a hobby: From comic art and camping, to woodworking and weaving
- Save travel inspiration: Outdoor adventures, family fun, road trips and more
- Find your style: Fashion, home decor, grooming tips and beauty inspiration
- Pin from your mobile browser: Save good things you find around the web

Figure 3.1: The Pinterest Android application description prior to sanitization.

pinterest visual bookmark tool help discov save creativ idea use pinterest make meal plan travel home improv project pinterest plan project home remodel garden redesign diy get creativ idea recip cook articl read gift buy way save money explor hobbi comic art camp woodwork weav save travel inspir outdoor adventur famili fun road trip find style fashion home decor groom tip beauti inspir pin mobil browser save good thing find around web

Figure 3.2: The Pinterest Android application description after sanitization.

which is not an actual category in Google Play [29]. Second, several of the Google Play categories are simply too broad and contain applications with stark differences with respect to their functionality. For example, alarm clock apps, keyboard apps, and flashlight apps can all be found in the *tools* category of Google Play. Next, Android applications are multifaceted and cannot be described by only one category. The popular Android application Twitter[†] is found in the *social* category on Google Play. However, it also provides the end user with the capability of sending direct messages to other Twitter users, which is a trait of applications found in the *communication* category of Google Play. Finally, using the categories on the Google Play marketplace violates the goal of making pDroid market independent.

To achieve market independence and give pDroid the capability to infer the many facets of an Android application, pDroid creates application categories using the well-known topic model, Latent-Dirichlet Allocation (LDA) [11]. Topic models are statistical models that can discover the semantic topics pervading a document set. After a topic model identifies the semantic topics, it can quantify the semantic differences between documents, allowing it to organize the otherwise unlabeled and unorganized dataset. The intuition behind LDA is that documents (application descriptions) exhibit traits from multiple topics (application categories). This intuition can be seen visually in the Android application description *Instagram*[‡] shown in Figure 3.3. In the description words related to *Communication* are highlighted in blue, words related to *Photos and Videos*, are highlighted in green, and the words highlighted in yellow are related to *Social Media*. If we took the time to label all words, a document-topic distrubution could be created showing how related a document is to a topic.

To capture this intuition, LDA assumes that documents in the collection arose from a set of topics, where a topic is defined as a distribution over words. To create topics, LDA uses a sampling algorithm such as Gibbs sampling [34] to find words

[†] https://play.google.com/store/apps/details?id=com.twitter.android
[‡] https://play.google.com/store/apps/details?id=com.Instagram.android

Share your photos and videos, and keep up with your friends and interests. *Instagram* is a simple way to capture and share the worlds moments. Follow your friends and family to see what theyre up to, and discover accounts from all over the world that are sharing things you love. Join the community of over 400 million people and express yourself by sharing photos and videos from your daywhether its your morning routine or the trip of a lifetime.
Use *Instagram* to:

- Edit and share photos and videos with filters and creative tools to change photo brightness, contrast and saturation, as well as shadows, highlights, perspective and more.
- Discover photos and videos you might like and follow new accounts in the Explore tab.
- Send private messages, photos, videos and posts from your feed directly to friends with *Instagram* Direct.
- Instantly share photos and videos on Facebook, Twitter, Tumblr and other social networks.

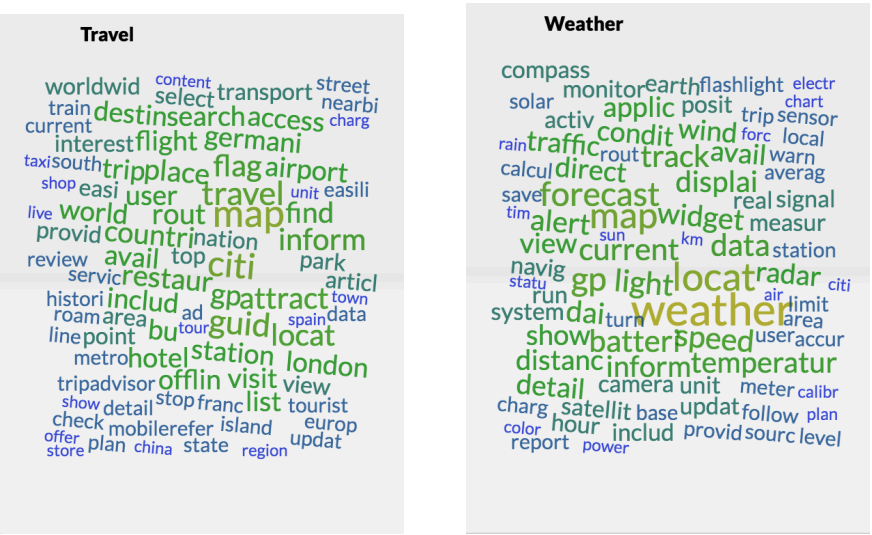
Figure 3.3: Application Description for the Android Application *Instagram*.

that frequently co-occur. Words that co-occur are assumed to be similar and will be assigned to the same topic. For example, after analyzing a set of application descriptions related to Weather, LDA created a topic containing the (stemmed) words "weather", "forecast," "temperatur", and "satellit." After analyzing applications related to Travel, LDA created a topic containing the words "flight", "visit", and "map." More words found in the Weather and Travel category can be found in Figure 3.4. Since LDA assumes that the documents in the collection arose from these topics, it can create a document-topic distribution, stating how likely a document originated from this topic, hence how similar it is to a topic.

pDroid uses the Mallet machine learning for language toolkit [42] to implement LDA. Mallet provides a Java-based implementation of LDA that uses an extremely fast and highly scalable implementation of Gibbs sampling and provides an efficient method for document-topic hyperparameter optimization. Following previous work, we train LDA using 30 topics [29]. Using thirty topics is based on the fact that Google Play uses 30 topics to categorize their applications, and it is possible that a more optimal amount of topics exist. Table 3.1 shows the top words in each topic after we trained LDA on application descriptions. After training, LDA can be used to infer the document-topic distributions for unseen application descriptions. When pDroid is given an unseen application description, it will first preprocess the description; then it will create a document-topic distribution using LDA.

The output of LDA does not provide a binary decision that a document belongs to a particular topic or does not belong in a topic. Instead, it provides a document-topic distribution, which is the probability that a document arose from a particular topic. An example of the document-topic distributions for the social media app for photo sharing *Instagram* and the racing game, *Mountain Climb Race 2*[§] are shown in Figure 3.5

[§]https://play.google.com/store/apps/details?id=com.awesomecargames.mountainclimbrace2



(a) Travel **(b)** Weather

Figure 3.4: Weather and Travel Topics Found By LDA

Table 3.1: Top words in each category.

Topic	Topic Name	**Top Stemmed Terms In Each Application Cluster.**
0	Language	languag word learn english german translat spanish french dictionari chines
1	Holidays and Religion	christma year holidai santa christian celebr gift polic islam tree
2	Cooking and Food	recip beer cake chicken appl cook chocol bake salad creams
3	Fitness and Diet	weight bodi diet exercis workout food lose yoga train
4	Fashion	girl beauti pictur sexi fashion cheerlead design hair high nail
5	Casino Games	slot machin card poker player coin casino spin bonu high
6	Fantasy Games	stori halloween world magic monster build adventur citi collect
7	Puzzle Games	puzzl level bubbl mode match challeng score classic player
8	Broadcasting	radio flag station countri channel world stream broadcast internet listen
9	Racing Games	race ball speed level jump control score challeng world mode
10	Reading	book question quiz answer read aikido logo test bibl reader
11	Photos and Videos	photo imag color share pictur facebook save friend effect
12	Weather	weather locat citi travel inform guid forecast rout attract find
13	Communication	version email googl user work contact permiss send internet requir
14	Action Games	weapon zombi enemi battl fight power world action shoot attack
15	Finance	calcul track manag data account market rate expens currenc list
16	Themes	theme launcher instal gold appli choos menu icon locker getjar
17	File and System	file player mobil control manag connect media wifi network secur
18	Music	music song album artist movi danc lyric youtub style record
19	Children	anim babi children learn fish memori child balloon sound mode
20	Anime	seri comic manga anim releas dragon charact film naruto imag
21	Ringtones	sound rington music alarm notif record sleep relax song
22	Legal	copyright page guid trademark cheat link owner develop unoffici trick
23	Inspirational	love make friend good find life great peopl give feel
24	Information	inform includ system provid medic product develop gener design refer
25	Sports	team footbal leagu player sport score basketbal soccer match club
26	Social Media	facebook twitter updat mobil share latest access find search friend
27	Utilities	widget batteri button option chang menu notif click displai model
28	Browsing	search galaxi samsung home icon delet creat notif easili shortcut
29	Data	home step press polici data background notif menu term privaci

Instagram was assigned to the following four topics:

- Topic 11 (Photos) with a probability of 60.2%.
- Topic 26 (Social Media) with a probability of 26.2%.
- Topic 9 (Racing Games) with a probability of 8.49%.
- Topic 24 (Information) witha probability of 4.80%.

Mountain Climb Race 2 was assigned to the following four topics:

- Topic 9 (Racing Games) with a probability of 90.4%.
- Topic 14 (Action Games) with a probability of 2.47%.
- Topic 23 (Inspirational) with a probability of 2.33%.
- Topic 7 (Puzzle Games) with a probability of .67%.

Figure 3.5: Topic Distrubutions for *Instagram* and *Mountain Climb Race 2*.

Originally, the final stage of clustering applications by alleged behavior was assigning an application to the four topics it was most highly distributed over. Therefore *Instagram* would be assigned to the "Photos", "Social Media", "Racing Games", and "Information" categories and *Mountain Climb Race 2* would be allocated to the categories "Racing Games", "Action Games", "Inspiration", and "Puzzle Games." The weakness of this approach was that it did not take into consideration the topic proportions for each application. Therefore, an application such as *Instagram* is given the same equality of determining the overall behavior as an application such as *Mountain Climb Race 2* in the "Racing Games" category.

The similarity between racing games and *Instagram* is difficult to understand. Game developers often describe the quality of the graphics found in their application in the description. Since graphics share a similarity between images and photos, LDA gives *Instagram* a slight distribution over the "Racing Games" category. To mitigate the issue of LDA inferring unwanted semantic similarities an additional clustering stage is implemented in pDroid which is discussed in Section 3.3.

3.3 Clustering Apps With Affinity Propagation

LDA does not assign an application description to one particular topic. Instead, it quantifies the semantic structure of the description by providing a document-topic distribution. A simple clustering approach would be to assign an application to its most related topic. However, many Android applications are multi-faceted and have traits rising from a variety of topics, so this approach would not capture the entire behavior of the application. Another possible approach would be to require an application description's proportion over a topic to meet a threshold before it is assigned to that cluster, but the ideal limit would be difficult to find and would be different for each cluster. Therefore, to cluster applications, pDroid uses Affinity Propagation (AP) [23], using the document-topic distributions as feature vectors. Unlike k-means, AP does not require the number of clusters to be determined before

running the algorithm. Instead, AP finds "exemplars" which are members of the dataset that are representative of clusters [23]. Affinity Propagation creates clusters by sending messages between pairs of samples until convergence. The messages sent between two points belong to one of two categories, the responsibility $r(i, k)$ or the availability $a(i, k)$. The responsibility is defined as the accumulated evidence that sample k should be the exemplar for sample i. The availability is the accumulated evidence that sample i should choose sample k to be its exemplar and takes into consideration the values for all other samples that k should be an exemplar. Therefore, exemplars are selected by samples if they are similar enough to many samples and chosen by many samples to be representative of themselves. To implement Affinity Propagation we use Python's Scikit-Learn: Machine Learning in Python Toolkit [47].

Chapter 4

Identifying Anomalous Behavior

After creating application clusters, the next goal of pDroid is to identify which applications are using anomalous dataflows within each cluster. In this chapter, we discuss how pDroid identifies abnormal *actual behavior* in each application cluster. Section 4.1 describes how pDroid extracts dataflows using FlowDroid [5]. Section 4.2 discusses Android dataflows, and Section 4.3 discusses the possible ways pDroid can represent dataflows. In Section 4.4 we discuss how pDroid quantifies the sensitivity of individual dataflows. Section 4.5 discusses how pDroid leverages application clusters and sensitivity scores to create an anomaly score for applications. Finally, we discuss how pDroid classifies an application as benign or malicious in Section 4.6.

4.1 Extracting Dataflows from Android Applications

To extract the permissions used by an Android application, one can just parse the application's manifest file. Obtaining the sensitive APIs from an application is also a relatively simple task, and can be completed using popular reverse engineering tools, such as *apktool* [63]. Unfortunately, extracting the dataflows found within an Android application is not as simple and requires dataflow analysis. pDroid uses static taint

analysis to obtain sensitive dataflows within an application. In taint analysis, sensitive information is first identified as a taint source, and data originating from a taint source is considered tainted. The tainted data is then tracked until it leaves the system through a tainted sink. Taint analysis can be completed statically or dynamically. In static taint analysis, the code (source, intermediate, or binary) is observed and reports any dataflows that may occur during the execution of the program. In dynamic taint analysis, the program is executed and reports the dataflows that occurred during runtime. In the static approach, an exhaustive search can be used to indicate all possible dataflows found within the code. Static analysis provides a complete picture of information flows found within applications. However, static analysis can report dataflows that are infeasible during execution time, leading to false positives. In contrast, during dynamic taint analysis, all dataflows reported are accurate, because they occurred during the runtime of the application. However, the results of dynamic analysis are bound to the test cases used to stimulate the application. In pDroid, we use the state-of-the-art static taint analysis tool, FlowDroid [5].

FlowDroid is a context-, flow-, field-, object-sensitive and lifecycle-aware static taint analysis tool for Android applications. Since Android applications do not have a standard *main()* entry point, FlowDroid creates a dummy *main()* method from the list of entry points found within the application. Next, FlowDroid uses the generated main method to create a call graph and inter-procedural control-flow graph (ICFG). FlowDroid then detects all sources of information that are reachable from the entry points. From these sources, FlowDroid applies taint analysis tracking by traversing the ICFG and reports all discovered flows from sources to sinks [24].

In order to achieve precise taint analysis, Flowdroid must take into consideration *flow-sensitivity*, *field-sensitivity*, *object-sensitivity*, and *context-sensitivity*. A taint-analysis technique is considered flow-sensitive if it takes into consideration the order of statements. In Figure 4.4.A, flow-sensitivity is required to identify that the sink on line 2 does not access the information coming from the source on line 3. In contrast, if the taint analysis technique were flow-insensitive, it would assume the source of

information on line 3 enters the sink on line 2, leading to a false positive. Context-sensitive taint analysis is required to track objects returned by method calls which are invoked with different input parameters. In Figure 4.4.B, context-sensitivity is necessary to distinguish that the sensitive information accessed on line 2 does not end up in the sink on line 3. Object-sensitivity is similar to context-sensitivity, and it takes into consideration the allocation site of the object to differentiate call sites. In Figure 4.4.C, object-sensitivity is required to distinguish that the sink on line 5 does not receive the information from the source argument in line 3. Finally, field-insensitive approaches merge taint information to the base object, while field-sensitive techniques treat all fields of a base object separately.

Applying static taint analysis on large Android applications comes with challenges. The major area of conflict described by Hammer et. al. is on one hand, the analysis should be correct and report all data leaks with few false negatives, but should be able to analyze real-world applications in a realistic amount of time [30]. To find an optimal trade-off between accuracy and runtime, we follow the work of [7], and configure FlowDroid with the following configurations:

- *No Flow Across Intents*: Android components communicate through a messaging object called an *Intent*. The most common use of Intents are to start an activity, start a service, or to deliver a broadcast. pDroid does not track dataflows through intents. Instead, all dataflows ending in intents are assigned the SUSI category, *INTENT*.
- *Explicit flows only*: FlowDroid currently focuses on explicit dataflows, and implicit flows caused by control-flow dependencies are ignored [25].
- *aliasflowins*. This option makes FlowDroid flow-insensitive, which improves the runtime for larger applications, but it may generate false positives.
- *aplength set to 3*. This sets the maximum access path length to 3, instead of the default of 5. Large access paths make analysis more precise but makes it more expensive.
- *No static fields* - This prevents tracking of static fields. This makes analysis faster, but it may miss potential data leaks.
- *No Layout Mode* - Input from Android GUI components are not taken into consideration as sources for dataflows.

```
int  x = 1;
sink(x);
x = source();
```

4.4.A

```
void contextExample(){
        String x =  foo("abc")
        String y = foo(source());
        sink(x);
}

String foo(String s){
        return s;
}
```

4.4.B

```
void objectExample(){
        Foo x = new Foo("bar");
        Foo y = new Foo(source());

        sink(x.getValue());
}

class Foo{
        String s;

        public foo(String inputString){
                s = inputString
        }

        public String getValue(){
                return s;
        }
}
```

4.4.C

Figure 4.4: (A) A flow-sensitive example. (B) A context-sensitive example. (C) A object-sensitive example.

With these configurations, we embed FlowDroid within pDroid to extract dataflows. After the dataflows are extracted, pDroid uses them to apply further analysis. We treat FlowDroid as a black box, and a more detailed description of its internal mechanisms can be found in the following literature, [5, 24, 25].

4.2 Dataflows in Android Applications

pDroid defines the actual behavior of an application by the set of dataflows found within the application. Applications running on an Android system are sandboxed, and applications can interact with the underlying system through an API interface provided by the Android platform. APIs that provide the application with resources are considered *sources* of data. APIs that export information from the application are called *sinks*. A dataflow is a tuple containing the signature of the source method and the signature of the sink method. For example, the API *LocationManager.getLastLocation()* returns the users last known location and is considered a source, and the API *URL.openConnection()*, which opens a connection to the referred URL is a sink.

While FlowDroid can provide taint tracking, it does not know which APIs in the Android interface are sources and sinks. Instead, the user must provide a list of sources and sinks as input. Creating a comprehensive list of sources and sinks for the Android platform is a challenging task due to the existing amount of APIs. For example, Androids version 4.2, contains 110,000 public methods, which makes manual classifications of sources and sinks infeasible [55]. One possible solution is permission maps [6, 8, 21], which identifies APIs in the Android interface that require a permission, and only considers those APIs as sensitive sources or sinks. However, this method only includes the subset of APIs that are protected by permissions, and the work of Rasthofer et. al. found that many sources or sinks did not require a permission [55]. For example, the *getNetworkOperatorName()* method in the *TelephonyManager* class returns the name of the network operator or carrier, but

does not require a permission. In pDroid, we use SUSI a fully automated machine-learning approach for identifying sources and sinks directly from Android source code [55]. SUSI does not consider only protected APIs but instead considers all APIs when discovering sources and sinks. SUSI is also fully automated, allowing the list of sources and sinks discovered by it to be updated when a new version of Android is released. SUSI also categorizes sources and sinks into human readable categories such as Location, Bluetooth, and Database. Finally, the list of SUSI categories was extended in the work of Avdiienko et. al. to include three new categories, intents, nonsensitive sink, and nonsensitive source. A list of all SUSI categories can be found in Table 4.1.

4.3 Representing Actual Behavior

FlowDroid provides pDroid with raw dataflows, which contains the full method signature for both the source and sink that created the dataflow. A method signature in Java contains the method name and the number and types of its parameters. Similar to [7], pDroid is configured to work on the following levels of granularity:

- **Method** This level of granularity is the finest, and uses the full method signature to define a source or sink. For example, *SmsManager.sendTextMessage(...)*
- **SUSI Category**, The most coarse grain method for representing a source or sink, would be to represent it by the SUSI category it was assigned to, such as *SMS_MMS*

In addition to this type of granularity, all nonsensitive sources and sinks, such as the device's display, are represented by the SUSI category, *No_Sensitive_Source* and *No_Sensitive_Sink*. We provide an example of representing the actual behavior at each level of granuality for the mobile application *EZ Clock & Weather Widget*[*] in Figure 4.5.

[*] https://play.google.com/store/apps/details?id=mobi.infolife.cwwidget&hl=en

Table 4.1: SUSI Source and Sink categorie

Sources	Sinks	Shared
HARDWARE INFO	PHONE_CONNECTION	AUDIO
UNIQUE IDENTIFIER	VOIP	SMS_MMS
LOCATION INFORMATION	PHONE_STATE	CONTACT_INFORMATION
NETWORK INFORMATION	EMAIL	CALENDAR_INFORMATION
ACCOUNT INFORMATION	BLUETOOTH	SYSTEM_SETTINGS
EMAIL INFORMATION	ACCOUNT_SETTINGS	IMAGE
FILE INFORMATION	SYNCHRONIZATION_DATA	BROWSER_INFORMATION
BLUETOOTH INFORMATION	NETWORK	NFC
VOIP INFORMATION	EMAIL_SETTINGS	
DATABASE INFORMATION	FILE	
PHONE INFORMATION	LOG	
CONTENT RESOLVER	INTENT	
NO SENSITIVE SOURCE	NO_SENSITIVE_SINK	

Method Granularity

NO_SENSITIVE_SOURCE $\rightsquigarrow$ Activity.setResult(int, content.Intent)
LocationManager.getLastKnownLocation(String) $\rightsquigarrow$ Log.i(String, String)
LocationManager.getLatitude() $\rightsquigarrow$ Log.i(String, String)
LocationManager.getLongitude() $\rightsquigarrow$ Log.i(String, String)

SUSI Granularity

NO_SENSITIVE_SOURCE $\rightsquigarrow$ INTENT
LOCATION_INFORMATION $\rightsquigarrow$ LOG
LOCATION_INFORMATION $\rightsquigarrow$ LOG
LOCATION_INFORMATION $\rightsquigarrow$ LOG

Figure 4.5: Dataflows in "EZ Clock & Weather Widget" using method, class, and SUSI granularity.

4.4 Creating Sensitivity Scores For Dataflows

pDroid uses all dataflows found within an application to define the behavior of an application. However, dataflows within an application are not equally as sensitive. For example, the source *TelephonyManager.getDeviceId()* returns the IMEI[†], if this information is sent to an unknown web server using the sink *URL.openconnection()* it is certainly a more sensitive dataflow then only writing the IMEI to a log file using the sink, *Log.i()*. Additionally, since the actual behavior varies between application clusters, the sensitivity of a dataflow should also depend on the application behavior found in a given cluster. To assign weights, we use an approah similar to the inverse-document frequency (idf) statistic found in term-frequency inverse-document-frequency (tf-idf) [57]; a well known statistic for detecting how important a word is to a document in the field of Information Retrieval. The inverse-document frequency reflects how important a word is to a document in a collection or corpus. Instead of considering words within a document we are considering how important a dataflow is to an application cluster. If many applications within an application cluster are using a particular dataflow, the dataflow should be considered less sensitive, and if the dataflow is uncommon in the cluster it should be considered a sensitive dataflow. To calculate the sensitivity weight for a dataflow, pDroid uses equation 4.1, where N is the amount of applications in application cluster c and a_d is the amount of applications in that cluster that use dataflow d.

$$W_{c,d} = \frac{N}{a_d} \tag{4.1}$$

4.5 Creating Anomaly Scores

The next goal of pDroid is to identify applications that have actual anomalous behavior. In pDroid, we consider an application to be exhibiting abnormal behavior

[†]International Mobile Station Equipment Identity

when it is using dataflows or combinations of dataflows that are uncommon in the application cluster it was assigned. Applications exhibiting anomalous behavior are considered outliers, and pDroid leverages outlier detection to identify applications that have unusual behavior. Specifically, pDroid uses Orca, a tool for detecting outliers based on their distance from their closest neighbors [9]. Orca uses the distance from a sample to its nearest neighbors to determine its similarity. The intuition is that if samples are close to other samples in the feature space, then the sample is most likely not an outlier. If an application is using dataflows that are extremely different from its nearest neighbors (most similar applications), then it probably exhibits unusual behavior and should raise suspicion. When using distance-based techniques for outlier detection, a distance metric must be defined. pDroid uses *weighted Euclidean distance*, shown in Eq. 4.2. Given a distance measure on a feature space, there are many different definitions of distance-based outliers [9]. Three popular definitions are:

1. Outliers are the example for which there are fewer than p other examples within distance d [35, 36]
2. Outliers are the top n examples whose distance to the kth neighbor is greatest [54]
3. Outliers are the top n examples whoses average distance to the k nearest neighbor is greatest [2, 18]

Similar to the reasoning in [37], the first definition of an outlier requires a maximum neighborhood, and does not provide an anomaly score for all applications. The second definition does not take into consideration the local density of samples. Therefore, we use the third definition to define an outlier in pDroid, where outliers are considered the n examples whose average distance to the k nearest neighbor is greatest. This definition provides pDroid with the most flexibility, allowing it to discover anomalies on a per-cluster basis.

$$d_{x,b} = \sum_{i=1}^{n} (w_i(x_i - b_i)^2)^{1/2}) \tag{4.2}$$

To use Orca, we convert the raw dataflows to feature vectors representing the dataflows an application uses. Since there are many ways to represent dataflows, there is also many ways to construct feature vectors for Orca to use. First, we take into consideration the granularity desired for analysis. pDroid is configured to provide anomaly scores using the two granularities discussed in Section 4.3. We found that *method* granularity was most suitable for providing anomaly scores. Next, an application may use a particular dataflow many times. We evaluated two different methods for representing the quantity of a dataflow. The first method took into consideration the number of occasions a dataflow was found within an application. Therefore, if dataflow d is found in application a v times, then $a_d = v$. The second approach is binary approach, and creates a feature vector based on if a dataflow was found within an application. If the application used dataflow d, then $a_d = 1$, if it did not then $a_d = 0$. Through our evaluation, we found that the binary approach outperformed the full approach. Therefore, for each application, pDroid creates a feature vector $\vec{a}$. If an application used a dataflow, d_i, then a_i is set to 1, and a_i is set to 0 if the application does not contain d_i. Since pDroid uses *weighted Euclidean distance*, pDroid must also create a weight vector, $\vec{w}$, for each application, where w_i represents the importance of d_i in the overall distance calculation. To create an application's weight vector, we use the per-cluster sensitivity scores discussed in Section 4.4. If an application is assigned to the application cluster c, then $w_i = W_{c,d_i}$. Additionally, we did not want to punish an application for not using a dataflow that one of its neighbors used. Therefore, if d_i is not found in an application, $w_i = 0$.

To create an anomaly score for an unknown app, a, that was assigned to application cluster c, pDroid runs Orca with k set to 5 and uses other applications assigned to c as the reference set (potential neighbors). The output of Orca will be an anomaly score representing the average distance from its five nearest neighbors. In Table 4.2 we provide the anomaly scores for an application cluster and malicous applications are highlighted in red.

Table 4.2: Anomaly Scores for an Application Cluster

air.com.mobigrow.canyouescape.apk	0.869	B
biz.mtoy.blockpuzzle.revolution.apk	7.23	B
com.PinballGame.apk	0.403	B
com.adwo.android.snake.apk0	79.725	M
com.bankey.candy.apk	14.703	B
com.bigduckgames.flow.apk	5.329	B
com.bitlogik.uconnect.apk0	34.555	M
com.bottleShootingGames.apk	6.719	B
com.camelgames.abnormalup.apk	38.54	M
com.cdroid.darts.apk	26.238	B
com.chenyx.tiltmazs.apk0	45.134	M
com.djinnworks.StickCliffDiving.lite.apk	1.765	B
com.game.BubbleShooter.apk	0.56	B
com.game.basketballshoot.apk	0.56	B
com.gp.jewels.apk1	54.284	M
com.hapogames.BubbleFarm.apk	0.909	B
com.icegame.fruitlink.apk	8.748	B
com.kiwifruitmobile.sudoku.apk	17.669	B
com.leagem.chesslive.apk	2.332	B
com.leagem.mahjong.apk	0.564	B
com.leftover.CoinDozer.apk	11.595	B
com.masshabit.squibble.free.apk0	44.33	M
com.mogo.threesameline.apk1	76.947	M
com.natenai.glowhockey2.apk	0.0	B
com.nix.game.mahjong.apk	2.949	B
com.oe.crazycorns.apk0	20.774	M
com.ps.yams.apk	71.856	M
com.threed.bowling.apk	6.097	B
com.wooga.diamonddash.apk	20.961	B

If an application receives a high anomaly score, pDroid can provide additional information about how each dataflow contributed to the overall anomaly score. For example, we used pDroid to inspect the malicious Android application, *com.camelgames.abnormal.apk* that belongs to the *BeanBot* Malware Family [33]. Table 4.3 shows the per-dataflow anomaly scores for each dataflow. The bolded dataflows represent the malicous payload that is transporting sensitive information to the app's command and control server. Since these dataflows are uncommon in other applications within the cluster, pDroid assigns these dataflows a high anomaly score. Additionally, this malware sample has the ability to stealthily send text messages in the background [33], which pDroid also detects.

4.6 Running pDroid as a Malware Classifier

Now that we have created anomaly scores for all applications, the final stage of pDroid uses supervised learning to classify applications as benign or malicious based on their anomaly scores. In addition to the anomaly scores, we found that applications with many dataflows naturally recieved higher anomaly scores. To address this issue we added an additional feature, the amount of unique dataflows found within an application. pDroid leverages the well known classification tool, Support Vector Machine (SVM) [13], to classify applications. Support Vector Machines are based on the concept of decision planes that define decision boundaries, where a decision plane is one that separates a set of objects that belong to different classes. To train our SVM, pDroid normalizes anomaly scores in each application cluster and aggegrates all applications to create a training set. The goal of traininig a Support Vector Machine is to find the separating hyperplane with the largest margin; the larger the margin, the better generalization of the classifier [16]. The basic intuition behind SVMs is to map the original input space into a feature space using a kernel, so that in the new feature space the data will be linearly separable. There are many types of kernels that can be used in SVM models, including linear, polynomial, radial basis function

Table 4.3: Anomaly Scores for each dataflow in *com.camelgames.abnormalup.apk*.

Dataflow	Anomaly Score
NO_SENSITIVE_SOURCE ⇝ ContextWrapper.openFileOutput()	0.429
NO_SENSITIVE_SOURCE ⇝ Log.w()	0.182
ContentResolver.query() ⇝ NO_SENSITIVE_SINK	0.429
ContentResolver.query() ⇝ URL.openConnection()	**5.0**
ConnectivityManager.getNetworkInfo() ⇝ NO_SENSITIVE_SINK	0.5
NetworkInfo.getExtraInfo() ⇝ NO_SENSITIVE_SINK	5.0
TelephonyManager.getDeviceId() ⇝ URL.openConnection()	**5.0**
TelephonyManager.getLine1Number() ⇝ URL.openConnection()	**5.0**
TelephonyManager.getSimSerialNumber() ⇝ URL.openConnection()	**5.0**
TelephonyManager.getSubscriberId() ⇝ NO_SENSITIVE_SINK	2.0
SmsManager.getDefault() ⇝ SmsManager.sendTextMessage()	**5.0**

(RBF), and sigmoid. The most commonly used kernel, and the one used in pDroid is the RBF kernel.

When training a SVM with a Gaussian Kernel, two parameters must be considered, the regularization parameter C and the kernel hyperparameter γ. The parameter C, controls the trade off between misclassificaiton and the simplicity of the decision surface. A low C value makes the decision smooth, while a high C aims at classifying training examples correctly. The γ parameter defines how far the influence of a single training example reaches. The larger gamma is, the closer other examples must be to be affected. Proper C and γ values are critical to an SVM's performance, and we leveraged Python Scikit-learn library's implementation of Grid Search to find the optimal paramaters.

Grid Search is a traditional hyperparameter optimization techniques that is simply an exhaustive sweep over a set of manually set of potential values. To perform grid search, one chooses a finite amount of reasonble values for the hyperparameters. In our case, we let C be equal to values of 1, 10, 100, and 1000 and γ to be values from .0001, .001, .01, and .01. Then we used grid search to train an SVM with each pair of (C, γ). Through our evaluation, we found that a C value of 100 and a γ value of .01 provided optimal results.

Chapter 5

Evaluation

In this chapter, we discuss the data collection, evaluation, and limitations of pDroid. Section 5.1 explains how we collected the dataset for evaluating pDroid. Next, in Section 5.2 we evaluated pDroid's ability to detect malware. In Section 5.3 we discuss how clustering applications improved performance. In Section 5.4 we evaluate our method for creating sensitivity scores increase, and Section 5.4 compares the different levels of granularity for dataflows. Also, this chapter also discusses the limitations of pDroid in Section 5.6.

5.1 Data Collection

To test pDroid, we leverage the dataset used to evaluate MudFlow [7]. This dataset contains the dataflows discovered from FlowDroid in 2,866 of the most popular Android apps from the Google Play Store, and dataflows found within 15,338 malware apps. The malware set came from two sources, VirusShare [56] and the Android Genome Project [65]. For simplicity, we will call this dataset, the MudFlow Dataset. Initially, this dataset lacked the necessary metadata for our analysis, such as the textual description. We used two approaches for obtaining the textual descriptions. First, we crawled the Google Play Store to check if the application still existed on the store. If this was the case, we saved the textual descriptions. If the application

was no longer on the Google Play Store, we checked if the application's description existed in the dataset used to evaluate Chabada [29], which we will call the Chabada Dataset. While we were not able to obtain textual descriptions for all 2,866 benign apps in the MudFlow Dataset, we were able to receive descriptions for 1562 benign apps.

It is often the case that malware repositories, such as VirusShare and the Android Genome Project only contain the .apk file for each sample and lack any metadata about the application. This makes obtaining malware metadata a more complicated process. To get malware, we first compared the malware samples found in the MudFlow and Chabada dataset for matches. This provided us with 118 malware samples containing both the textual description and the dataflows. To increase our dataset, we leveraged a similar technique found in [29]. Many of the samples found within the Android Genome Project are repackaged versions of benign apps. These applications provide the same functionality as the original application but have the malicious payload injected into them. Therefore, we installed the malware onto an Android emulator to evaluate the expected behavior of the application; then we searched the Google Play Store for an application that would provide similar functionality and used its textual description to represent the malicious app's alleged behavior. Using this method, we were able to obtain 243 malware samples in overall.

We evaluated pDroid using 1562 benign apps and 243 malware samples.

5.2 PDroid for Malware Detection

Our first goal is to evaluate how effective pDroid can correctly classify applications as benign or malicous. To evaluate our system, we use standard machine learning evaulation approaches [16]. We define a true positive, true negative, false positive, and false negative as the followiing:

- **True Positive (TP)** - pDroid correctly classifiers a malicious app as malware.
- **True Negative (TN)** - pDroid correctly classifies a benign application as benign.
- **False Positive (FP)** - pDroid missclassifies a benign application as malicous.
- **False Negative (FN)** - pDroid missclasssifies a malicous application as benign.

To evaluate pDroid's ability to detect malware, we take into consideration the True Positive Rate (TPR), True Negative Rate (TNR), and Accuracy. Since our dataset is extremely unbalanced (contains more benign samples then malware) the standard method for calculating accuracy, shown in Eq. 5.1, would not be ideal. Instead, we use the geometric accuracy, shown in Eq. 5.2. To evaluate our system, we use *stratified 10-fold cross validation*. In k-fold cross-validation, the original dataset is randomly partitioned into k equal size partitions ("folds"). One fold is held out and is considered the testing set. The other $k-1$ partitions are used for training the classifier. This approach is repeated k times, where each time a different partition is held out for testing. Stratified k-fold cross-validation extends this approach by preserving the percentage of samples for each class in each fold [47]. Therefore, we run pDroid ten times, each time with a different set of training and testing samples. For each run, we report the TPR, TNR, and geometric accuracy. The results for each run can be found in Table 5.1. pDroid's geometric acccuracy was 93.5 % with a false positive rate of 4.9%.

pDroid correctly classified 91.4% of malware, with a false positive rate of 4.9%.

$$accuracy = \frac{TP + TN}{TP + FN + TN + FP} \tag{5.1}$$

$$g = \sqrt{TP \times TN} \tag{5.2}$$

Table 5.1: Cross Validation Results for pDroid.

run	**TNR (%)**	**TPR (%)**	**Geometric Accuracy** (%)
0	96	91	93.5
1	96	92	94.0
2	94	94	94.0
3	94	90	92.0
4	96	93	94.5
5	95	92	93.5
6	95	92	93.5
7	94	92	93.0
8	97	92	94.5
9	94	91	92.5
Avg	95.1	91.4	93.5

Table 5.2: Comparison of Clustering Techniques.

Clustering Technique	**TNR** (%)	**TPR** (%)	**Geometric Accuracy** (%)
Affinity Propagation	95.1	91.9	93.5
K Means (30 clusters)	93.2	91.0	92.45
No Clustering	88.0	90	89

Table 5.3: Evaluation of Sensitivity Scores.

Sensitivity	**TNR** (%)	**TPR** (%)	**Geometric Accuracy** (%)
Sensitivity Scores	95.1	91.5	93.1
No Sensitivity Scores	90.1	91.0	90.2

5.3 Evaluating the use of application clusters

Next we wanted to evaluate what improvements were provided by clustering applications before creating anomaly scores. we evaluated the performance of pDroid using no clusters, k-means clustering, and affinity propagation. Table 5.2 shows the results for each technique. We found that application clustering improved the false positive rate by 7.1%.

> Application clustering reduces the false positive rate by 7.1%.

5.4 Evaluating Dataflow Sensitivity Scores

For each dataflow, pDroid assigned a sensitivity score that quantifies the sensitivity of a given dataflow. To evaluate how useful sensitivity score we ran pDroid assuming all dataflows were equally as sensitive, and the results are shown in Table 5.3. We found that using sensitivity scores improved the TNR by 5.0%.

Table 5.4: Evaluation of Granularity

Granularity	**TNR**(%)	**TPR** (%)	**Geometric Accuracy** (%)
Signatures	93.4	91.4	92.4
SUSI Categories	87.7	85.8	86.7

5.5 Using Varying levels of granualrity

Next, we use SUSI categories to represent dataflows. Despite being a more coarse-grained approach, using SUSI categories performed well and received a geometric accuracy of 86.7%.

5.6 Limitations

There are several limitations with pDroid. As with other static analysis systems [26, 39, 40] on Android, pDroid cannot detect privacy disclosures caused by Java reflection, code obfuscation, or dynamic code loading. pDroid is also susceptible to the pollution attack described in [39], where a malicous developer intentially creates many malicous applications with similar textual descriptions, causing pDroid to create a "malicous cluster" where malcious dataflows are common and therefore will go undetected. A simple solution to this issue would be to only compare applications that are developed by different developers. However, this could be easily by passed by registering the malicious applications under different names.

Chapter 6

Conclusions

This chapter concludes this Master Thesis by discussing the Future Work and Conclusion. Section 6.1 discusses the future work and potential improvements that could be made to pDroid. Section Section 6.2

6.1 Future Work

Many frameworks can detect malware, but pDroid unique method of comparing most similar applications should allow it to be a successful tool in identifying "grayware" applications. These type of applications generally are not considered malicious but are not being fully transparent about the application's full ability. In future work, we plan on adapting pDroid to focus on detecting grayware instead of only malware. Additionally, we want to use pDroid to provide security reports to end users, who can use them to interpret the potential security risks of an application.

While pDroid was able to detect malware at a high rate, there is room for improvement. First, we only consider the textual description to be the alleged behavior of an application. However, the amount of downloads, size, and recommend app's, and could provide more insight on the application's behavior. Next, pDroid currently assumes that advertising frameworks should be trusted and does not take into consideration the dataflows within them. However, they are most likely leaking

sensitive data about the end user. This type of behavior should be stated in the textual description. pDroid uses the amount of dataflows found within an application to assist in classifying an app as benign or malicious. A malicous application could easily trick pDroid by adding an excessive amount of benign dataflows into their application, which would cause pDroid to misclassify it. Therefore, a new technique for comparing applications of similar sizes needs to be developed.

6.2 Conclusion

In this master thesis we presented pDroid, a framework for comparing the textual description of an application to the internal dataflows of an application. Unlike previous approaches, pDroid uses a more fine-grained approach to capture an application's actual behavior. pDroid correctly identified 91.4% of malware with a false positive rate of 4.9%.

Bibliography

[1] Android. *Android Security*. Accessed: 5-24-2016. URL: https://source.android.com/security/ (cit. on pp. 6, 8, 9).

[2] Fabrizio Angiulli and Clara Pizzuti. "Fast outlier detection in high dimensional spaces". In: *PKDD*. Vol. 2. Springer. 2002, pp. 15–26 (cit. on p. 39).

[3] *App Store Review Guidelines*. Accessed: 6-7-2016. 2016. URL: https://developer.apple.com/app-store/review/guidelines/ (cit. on p. 10).

[4] AppBrain. *Number of available applications in the Google Play Store from December 2009 to February 2016. In Statista - The Statistics Portal*. Accessed: 5-24-2016. URL: http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ (cit. on p. 4).

[5] Steven Arzt et al. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps". In: *ACM SIGPLAN Notices*. Vol. 49. 6. ACM. 2014, pp. 259–269 (cit. on pp. 30, 31, 34).

[6] Kathy Wain Yee Au et al. "Pscout: analyzing the android permission specification". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 217–228. URL: http://dl.acm.org/citation.cfm?id=2382222 (visited on 03/17/2015) (cit. on pp. 12, 14, 15, 34).

[7] Vitalii Avdiienko et al. "Mining apps for abnormal usage of sensitive data". In: *2015 International Conference on Software Engineering (ICSE)*. 2015. URL: https://www.st.cs.uni-saarland.de/appmining/mudflow/icse2015-mudflow.pdf (visited on 06/09/2015) (cit. on pp. 32, 35, 45).

[8] Alexandre Bartel et al. "Automatically securing permission-based software by reducing the attack surface: An application to android". In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2012, pp. 274–277 (cit. on p. 34).

[9] Stephen D Bay and Mark Schwabacher. "Mining distance-based outliers in near linear time with randomization and a simple pruning rule". In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM. 2003, pp. 29–38 (cit. on p. 39).

[10] Steven Bird. "NLTK: the natural language toolkit". In: *Proceedings of the COLING/ACL on Interactive presentation sessions*. Association for Computational Linguistics. 2006, pp. 69–72 (cit. on p. 18).

[11] David M. Blei, Andrew Y. Ng, and Michael I. Jordan. "Latent dirichlet allocation". In: *the Journal of machine Learning research* 3 (2003), pp. 993–1022. URL: http://dl.acm.org/citation.cfm?id=944937 (visited on 07/01/2014) (cit. on pp. 17, 22).

[12] 9 Apps Business. *About US*. Accessed : 6-8-2016. 2016. URL: http://business.9apps.com/about/ (cit. on p. 11).

[13] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine learning* 20.3 (1995), pp. 273–297 (cit. on p. 42).

[14] Cathy De Rosa et al. *Perceptions of libraries, 2010: Context and community: A report to the OCLC Membership*. OCLC, 2011 (cit. on p. 9).

[15] Jason Riesa Dick Sites Andrew Hayden. *Chromium Compact Language Detector 2*. 2015 (cit. on p. 18).

[16] Richard O Duda, Peter E Hart, and David G Stork. *Pattern classification*. John Wiley & Sons, 2012 (cit. on pp. 42, 46).

[17] David Ehringer. "The dalvik virtual machine architecture". In: (2010) (cit. on p. 5).

[18] Eleazar Eskin et al. "A geometric framework for unsupervised anomaly detection". In: *Applications of data mining in computer security*. Springer, 2002, pp. 77–101 (cit. on p. 39).

[19] Adrienne Porter Felt, Serge Egelman, and David Wagner. "I've got 99 problems, but vibration ain't one: a survey of smartphone users' concerns". In: *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2012, pp. 33–44 (cit. on p. 1).

[20] Adrienne Porter Felt et al. "A survey of mobile malware in the wild". In: *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM. 2011, pp. 3–14 (cit. on p. 11).

[21] Adrienne Porter Felt et al. "Android permissions demystified". In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011, pp. 627–638 (cit. on pp. 15, 34).

[22] Adrienne Porter Felt et al. "Android permissions: User attention, comprehension, and behavior". In: *Proceedings of the Eighth Symposium on Usable Privacy and Security*. ACM. 2012, p. 3 (cit. on p. 1).

[23] Brendan J Frey and Delbert Dueck. "Clustering by passing messages between data points". In: *science* 315.5814 (2007), pp. 972–976 (cit. on pp. 17, 28, 29).

[24] Christian Fritz. "Flowdroid: A precise and scalable data flow analysis for android". PhD thesis. 2013 (cit. on pp. 31, 34).

[25] Christian Fritz et al. "Highly precise taint analysis for Android applications". In: *EC SPRIDE, TU Darmstadt, Tech. Rep* (2013) (cit. on pp. 32, 34).

[26] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. "Scandroid: Automated security certification of android". In: (2009) (cit. on p. 50).

[27] Evgeniy Gabrilovich and Shaul Markovitch. "Computing semantic relatedness using wikipedia-based explicit semantic analysis." In: *IJCAI*. Vol. 7. 2007, pp. 1606–1611 (cit. on p. 14).

[28] Gartner. *Gartner Says Worldwide Smartphone Sales Grew 9.7 Percent in Fourth Quarter of 2015*. Accessed: 5-24-2016. 2016. URL: https://www.gartner.com/newsroom/id/3215217 (cit. on pp. 4, 11).

[29] Alessandra Gorla et al. “Checking app behavior against app descriptions”. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1025–1035. URL: http://dl.acm.org/citation.cfm?id=2568276 (visited on 06/10/2015) (cit. on pp. 2, 15, 22, 24, 46).

[30] Christian Hammer and Gregor Snelting. “Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs”. In: *International Journal of Information Security* 8.6 (2009), pp. 399–422 (cit. on p. 32).

[31] Luis E Hestres. “App neutrality: Apples app store and freedom of expression online”. In: *Hestres, LE (2013). App Neutrality: Apples App Store and Freedom of Expression Online. International Journal of Communication* 7 (2013), pp. 1265–1280 (cit. on p. 11).

[32] Mikko Hypponen. “Malware goes mobile”. In: *Scientific American* 295.5 (2006), pp. 70–77 (cit. on pp. 9, 10).

[33] X Jiang. *Security alert: New beanbot sms trojan discovered* (cit. on p. 42).

[34] Chang-Jin Kim, Charles R Nelson, et al. *State-space models with regime switching: classical and Gibbs-sampling approaches with applications*. Vol. 2. MIT press Cambridge, MA, 1999 (cit. on p. 22).

[35] Edwin M Knorr and Raymond T Ng. “Finding intensional knowledge of distance-based outliers”. In: *VLDB*. Vol. 99. 1999, pp. 211–222 (cit. on p. 39).

[36] Edwin M Knorr, Raymond T Ng, and Vladimir Tucakov. “Distance-based outliers: algorithms and applications”. In: *The VLDB JournalThe International Journal on Very Large Data Bases* 8.3-4 (2000), pp. 237–253 (cit. on p. 39).

[37] Konstantin Kuznetsova et al. “Mining Android Apps for Anomalies”. In: () (cit. on p. 39).

[38] Hiroshi Lockheimer. *Android and Security*. Accessed: 6-7-2016. URL: http://googlemobile.blogspot.de/2012/02/android-and-security.html (cit. on p. 11).

[39] Kangjie Lu et al. "Checking More and Alerting Less: Detecting Privacy Leakages via Enhanced Data-flow Analysis and Peer Voting." In: *NDSS*. 2015 (cit. on pp. 15, 50).

[40] Long Lu et al. "Chex: statically vetting android apps for component hijacking vulnerabilities". In: *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM. 2012, pp. 229–240 (cit. on p. 50).

[41] Siqi Ma et al. "Active Semi-supervised Approach for Checking App Behavior against Its Description". In: *Computer Software and Applications Conference (COMPSAC), 2015 IEEE 39th Annual*. Vol. 2. IEEE. 2015, pp. 179–184 (cit. on p. 2).

[42] Andrew Kachites McCallum. "Mallet: A machine learning for language toolkit". In: (2002) (cit. on p. 24).

[43] MediaPost. *Number of Smartphone Users in The United States from 2010 to 2019 (in Millions)**. Accessed: 6-7-2016. URL: http://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/ (cit. on p. 9).

[44] C Miller. "Inside ios code signing". In: *Symposium on Security for Asia Network (SyScan)*. 2011 (cit. on p. 10).

[45] Cheeta Mobile. *Android App Stores Become Significant Source for Malware*. Accessed : 6-8-2016. 2016. URL: http://www.cmcm.com/blog/en/security/2016-01-20/925.html (cit. on p. 13).

[46] Rahul Pandita et al. "WHYPER: Towards Automating Risk Assessment of Mobile Applications." In: *USENIX Security*. Vol. 13. 2013. URL: http://hibou.

cs.wpi.edu/~kven/courses/CS4401-C15/papers/Whysper.pdf (visited on 03/17/2015) (cit. on pp. 2, 12).

[47] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830 (cit. on pp. 29, 47).

[48] Nicholas J Percoco and Sean Schulte. "Adventures in bouncerland". In: *Black Hat USA* (2012) (cit. on p. 11).

[49] Sarah Perez. "App Submissions On Google Play Now Reviewed By Staff, Will Include Age-Based Ratings". In: (2015). URL: http://techcrunch.com/2015/03/17/app-submissions-on-google-play-now-reviewed-by-staff-will-include-age-based-ratings/ (cit. on p. 12).

[50] Martin F Porter. "An algorithm for suffix stripping". In: *Program* 14.3 (1980), pp. 130–137 (cit. on p. 19).

[51] PulseSecure. "2015 Mobile Threat Report". In: (2015). Accessed : 6-8-2016. URL: https://www.pulsesecure.net/lp/mobile-threat-report-2014/ (cit. on p. 12).

[52] Zhengyang Qu et al. "AutoCog: Measuring the Description-to-permission Fidelity in Android Applications". en. In: ACM Press, 2014, pp. 1354–1365. ISBN: 9781450329576. DOI: 10.1145/2660267.2660287. URL: http://dl.acm.org/citation.cfm?doid=2660267.2660287 (visited on 06/10/2015) (cit. on pp. 1, 2, 14, 15).

[53] Daniel Ramage and E Rosen. *Stanford topic modeling toolbox*. 2011 (cit. on p. 18).

[54] Sridhar Ramaswamy, Rajeev Rastogi, and Kyuseok Shim. "Efficient algorithms for mining outliers from large data sets". In: *ACM SIGMOD Record*. Vol. 29. 2. ACM. 2000, pp. 427–438 (cit. on p. 39).

[55] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. "A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks." In: *NDSS*. 2014 (cit. on pp. 34, 35).

[56] J-Michael Roberts. *VirusShare. com*. 2014 (cit. on p. 45).

[57] Gerard Salton and Christopher Buckley. "Term-weighting approaches in automatic text retrieval". In: *Information processing & management* 24.5 (1988), pp. 513–523 (cit. on p. 38).

[58] Neil Smyth. *Android 4 App Development Essentials*. Ed. by Payload Media. Accessed: 5-24-2016. 2014. URL: http://www.techotopia.com/index.php/Android_4_App_Development_Essentials (cit. on p. 7).

[59] Ryan Stevens et al. "Asking for (and about) permissions used by android apps". In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. IEEE Press. 2013, pp. 31–40 (cit. on p. 15).

[60] *Threat Description: Bluetooth-worms:symbos/Cabir*. Accessed: 6-7-2016 (cit. on p. 9).

[61] Roman Unucheck and Chebyshev Victor. *Mobile malware evolution 2015*. Accessed: 6-7-2016 (cit. on p. 10).

[62] Tielei Wang et al. "Jekyll on iOS: When Benign Apps Become Evil." In: *Usenix Security*. Vol. 13. 2013 (cit. on p. 11).

[63] R Winsniewski. *Android–apktool: A tool for reverse engineering android apk files*. 2012 (cit. on p. 30).

[64] Yingyuan Yang, J Stella Sun, and Michael W Berry. "APPIC: Finding The Hidden Scene Behind Description Files for Android Apps". In: () (cit. on p. 2).

[65] Yajin Zhou and Xuxian Jiang. "Dissecting android malware: Characterization and evolution". In: *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE. 2012, pp. 95–109 (cit. on p. 45).

Vita

Joe Allen is a Master's candidate at the Department of Electrical Engineering and Computer Science at the University of Tennessee. His research interests are in the field of Android security, cybersecurity, machine learning, and natural language processing. Joe received his Bachelor's degree in Computer Engineering from the University of Tennessee, Knoxville in the Fall of 2014. Joe has worked as a graduate teaching and research assistant under the supervision of Dr. Jinyuan Sun, since 2014.