ANDROID BEHIND THE SCENES: REVEALING HIDDEN MALWARE WITH ANDROMEDA

BY

ROBERT JOSEPH MARSAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2013

Urbana, Illinois

Adviser:

Professor Roy H. Campbell

Abstract

With the unprecedented growth in the diversity of digital distribution platforms there has been an increasing concern about privacy of the content being produced by the different types of applications users are installing across a variety of mobile platforms. The mobile devices average users own generally have inbuilt hardware interfaces capable of gathering different types of rich information including temperature, accelerometer, as well as personal identifiable information such as phone numbers, personal communication messages, and location information. Some digital distribution platforms, like Google's Play Store (GPStore), put into effect a permission based security model where users are prompted with a list of permissions they must accept in order to download and install and application. In recent years many researchers have presented attacks compromising the previous security model using different types of techniques including malicious applications such as malware and trojans.

To better understand mobile malware, we introduce the concept of the User-App Agreement (UAA) - a conceptual framework for a user consenting and trusting specific actions an app may take. Using UAA we examine the Android Permission system with Android Census - a comprehensive app metadata database. We conclude the main shortcoming lies in the Permission system's lack of addressing context and use, presenting an opportunity for Info Theft Malware - malicious software that profits off of user's personal information. Finally, we present Android Malware Evaluation Detection and Analysis (AndroMEDA), an Android Security Extension which forms a novel feedback loop, providing users with a method for understanding the context and use of actions an app performs, thus allowing them to identify suspicious behavior that violates users' trust.

To my

Parents, Lisa and Mark

Acknowledgments

I would like to express my most sincere gratitude towards my advisor Roy H. Campbell for his continuous guidance and support of my work, as well as Lynette Lubben for her patience. Roy and Lynette opened many doors for my research, and I could not be more appreciative.

I cannot find the words to express my gratitude towards Alejandro Gutierrez for his tireless support and assistance, and unwavering confidence and dedication. My research would not be what it is today without his contributions and mentorship. The man's dedication is under appreciated, and I hope he sleeps sometime.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Thesis Statement: Using a novel feedback loop, we provide users with a method for understanding the context and use of actions a mobile app performs, thus allowing them to identify suspicious behavior that violates users' trust.

The rise of smartphones in the last decade years has been unprecedented. Since the launch of the Apple iPhone in 2007, there are now almost 1 billion smartphone users in the world[39]. These new devices marked an unprecedented shift in our relationship with computers, becoming the center point for many personal endeavors, and superseding almost all previous computing devices from cell phones, to cameras, to GPS devices, and to most uses of a desktop PC[31]. Smartphones continue to become the focal point of almost all personal computing, and consequently the operating systems they run become more important and powerful.

1.1 Contributions

In this thesis, we highlight 4 key areas:

- *User-App Agreement*: First, we discuss the challenges of addressing modern mobile malware, and the shortcomings of the Android security model. We introduce the User-App Agreement (UAA), a way of conceptualizing the trust a user has in actions an applications may perform, as a key component in identifying malicious behavior.
- *Android Census*: Second, we use a novel dataset, Android Census, to examine the state of Android permissions. We find Android permissions correlate with expected use, but key examples are shown of less than legitimate use. Using a comprehensive set of malware, we cross examine how the permissions of malware compares with the Android Census. We conclude that malware that targets the user's personal information is the most difficult to detect using this static analysis.
- *IncognitoWare*: Third, we address the shortcomings of the current malware datasets available to academia, and introduce a new dataset of IncognitoWare. This dataset is more representative of current trends in malware, and proves to be a great challenge to detect.

- *AndroMEDA*: Finally, we introduce Android Malware Evaluation Detection and Analysis (AndroMEDA), a set of Android extensions, and a companion app, built *off of* the premise of the User-App Agreement. By giving the user more information on the context and use of sensitive system actions, they can evaluate whether they trust those actions, and ultimately whether the app is acting maliciously or not.

Chapter 2

Background & Motivation

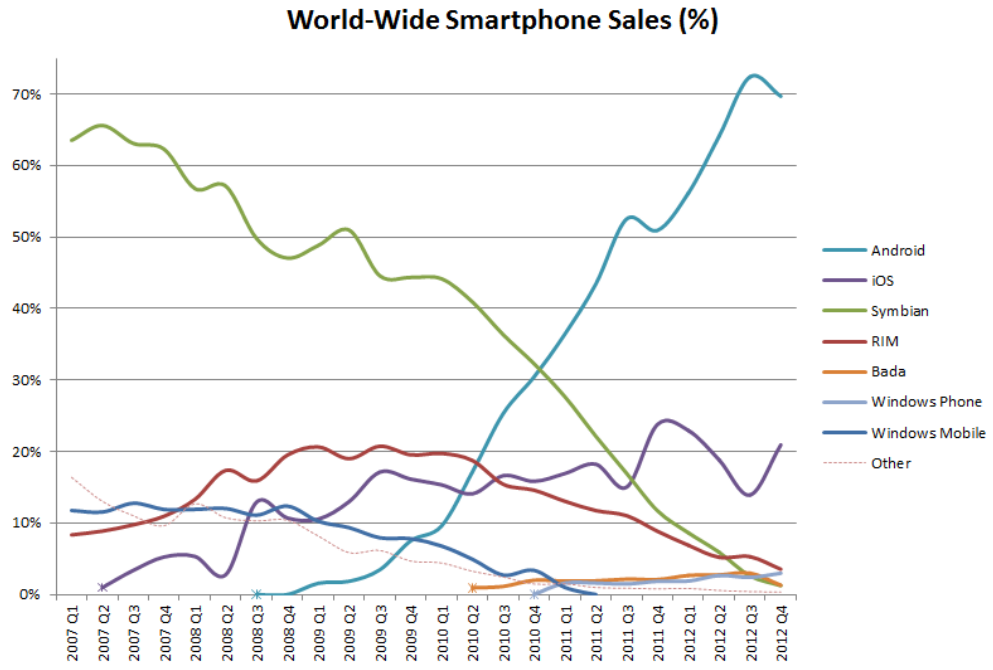

Figure 2.1: Worldwide Market Share of various mobile OSs - from [61] and [25]

Mobile OSs, like the PC operating systems of the 1990s, have a few major players that wield the most influence, as seen in Figure 2.1. The two largest operating systems in the mobile area are Android and iOS. Apple's iOS, made exclusively for the Apple iPhone and iPad, as of the end of 2012, runs on over 20%[25] of all smartphones globally. Google's Android, released as an open source OS, has many different hardware manufacturers, Samsung, LG, HTC, Motorola, and many more. It currently runs the majority of smartphones globally, with 70%[25] marketshare. Some of the less popular, but still significant mobile operating systems are Windows Phone, with 3%, and Blackberry, with 3.5%[25] .

2.1 iOS

Apple released the iPhone in 2007. “Entry into mobile phones might have been a risky move for Apple. The industry was dominated by Nokia, Motorola, and Samsung, with roughly 60% market share.”[66]. However, “the Apple iPhone was a huge success. Considered by Time magazine the invention of the year 2007 (Invention of the year: the iPhone, Time 2007), it completely changed the mobile phones industry dynamics.”[48]. Apple’s iPhone and iOS were novel thanks to its touch friendly and intuitive OS, and their digital distribution platform, the App Store[66], and in 2012, Apple sold over 130 Million iOS devices[25].

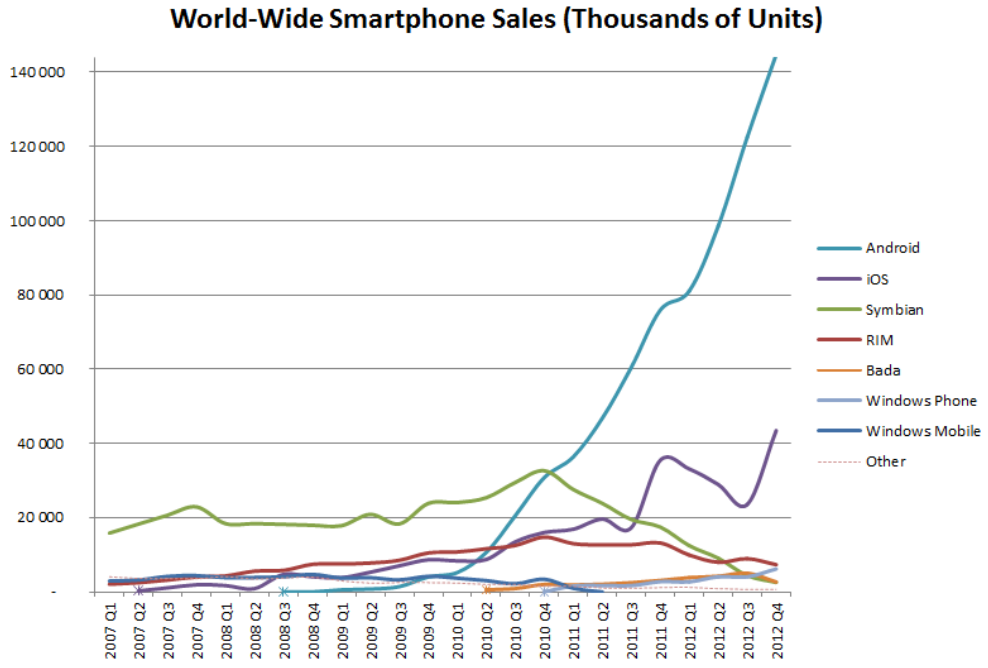

Figure 2.2: Worldwide Sales of various mobile OSs - from [62] and [25]

2.2 Android

Started in 2003 by Andy Rubin and Android Inc. (previously the makers of the T-Mobile Sidekick), Android was acquired by Google Inc. in 2005[20]. “Android was built from the ground-up to enable developers to create compelling mobile applications that take full advantage of all a handset has to offer. It was built to be truly open. For example, an application can call upon any of the phone’s core functionality such as making calls, sending text messages, or using the camera, allowing developers to create richer and more cohesive experiences for users”[45]. Since it’s initial release in 2007[44], Android has skyrocketed to the most used mobile OS in the world, with over 70% marketshare,

and 144 Million Android devices being sold in Q4 of 2012 alone[25] - more than Apple had the entire year, as seen in Figure 2.2.

2.3 Goals of Mobile OSs

For all these mobile OSs, they share many common goals and challenges. The diversity of hardware that smartphones were designed to replace, along other constraints and features, requires a mobile OS that's designed from the ground up to deal with many different challenges than the typical PC OS. Some of the main design challenges for a mobile OS are:

- Small memory footprint, battery conscious, and other resource constrictions
- Access to a wide variety of personally identifiable information (PII)
- Access a wide array of hardware

In order to effectively enforce rules on battery consumption, low-latency UI, and personally identifiable information, a new security model was created, centered around the concept of the "App".

2.4 The "App" and Sandboxing

In the mobile world, "Apps" are isolated and sandboxed programs, generally designed with one singular purpose. They lack dependencies, and generally are not as privileges as system software for performing many tasks. The mechanisms for accessing functionality outside of their sandbox is enforced by a set of policies the system holds, specific to that app. On some platforms, like iOS, only one app may run at any given time, and background computation is virtually non-existent (with some exceptions)[1], along with many other restrictions. On Android and other platforms, many more features are available to apps, but in all cases, the "app" lifecycle is well defined and controlled by the system much more than on a PC OS.

There are various reasons for the tight sandboxing of mobile apps. Power and resource consumption are certainly a factor - mobile OSs generally reserve the right to kill apps if they attempt to allocate too much memory. Controlling access to hardware also helps in this: allowing apps to keep the phone awake could easily drain battery. However, another reason for sandboxing, and arguably more important, is protecting Personally Identifiable Information.

[1] Minor amounts of computation can be done to compute background audio, and other isolated background tasks.

2.4.1 Personally Identifiable Information

Personally Identifiable Information (PII), as defined by the National Institute of Standards and Technology, is "any information about an individual maintained by an agency, including (1) any information that can be used to distinguish or trace an individuals identity... and (2) any other information that is linked or linkable to an individual" [38]. Mobile devices, having blended cameras, cell phones, GPS devices, and PCs into one device, have an extremely diverse amount of PII, from phone numbers, to contacts, to location history, to bank account numbers and pictures. For many of these datasets, mobile OSs actually organize them into databases with the intention of allowing 3rd parties access to them. Contact lists, SMS, Photographs and location history are available to apps on virtually every mobile platform in some official way. This is a driving motivation for a greatly improved security model for mobile OSs: controlling 3rd party software's access to PII.

2.4.2 Digital Distribution Platform

The final major difference between mobile OSs and PC OSs is the distribution of code. No mobile OS allows 3rd party code to be ran outside of the sandbox, and all of them require the user's consent before installing an app. All apps must be signed, and in general, there is 1 main distribution channel for all apps on a mobile OS. This tightly controlled distribution both aids in security, as well as controls the ecosystem around that mobile OS.

2.4.3 Apple's App Store

The first major digital distribution platform for mobile apps was Apple's App Store[11]. Its model has been repeated by almost all major mobile app distribution platforms. The basic premise is simple: developers sign up to the app store, pay a fee (usually yearly), and submit fully-finished apps. A reviewer runs the app in a monitored sandbox, watches for unusual behavior, checks for stability and usability, and approves it. Once the app has been approved, it's released onto the app store, at which time anyone can download it. The approval process, as well as the high monetary fee, act as a way to ensure only safe and high-quality apps are available for that platform. In this type of platform, typically no apps may be installed from other sources. On iOS, initially this was the main method of security: if the app passed the inspection, it was acknowledged as safe and virtually unmonitored unless someone noticed something unusual and reported it. However, in recent years, after certain incidents (see section 4.6.1), apps still must request permission from the user to perform certain tasks.

2.4.4 Android Permissions

Android's distribution platform takes a different approach, and at its core is also Android's security model: The Permission System (see section 3). Android Apps declare when they are packaged what capabilities they will use, and the user reviews them at install time. If the user approves the app, it may use the requested capabilities whenever it wants: little restrictions are placed otherwise. With this barrier in mind, the Google Play Store (formerly known as the Android Market), or GPStore, opts for an alternate model to iOS, where the developer pays a smaller fee, and apps go through no formal approval process. After an app's submission, it's immediately released into the wild for users to download and run. The assumption Android uses is that the metadata the GPStore provides: App name, Developer Name, Description, Reviews and Ratings, are enough for the user to determine if the app should be trusted with the permission set it's given (see Figure 2.3). In fact, Android even allows the device to accept apps from 3rd party sources, a practice known as "sideloading", although it's disabled by default. This has spawned a large number of 3rd party app sources, all of which rely on the Permission system for user protection.

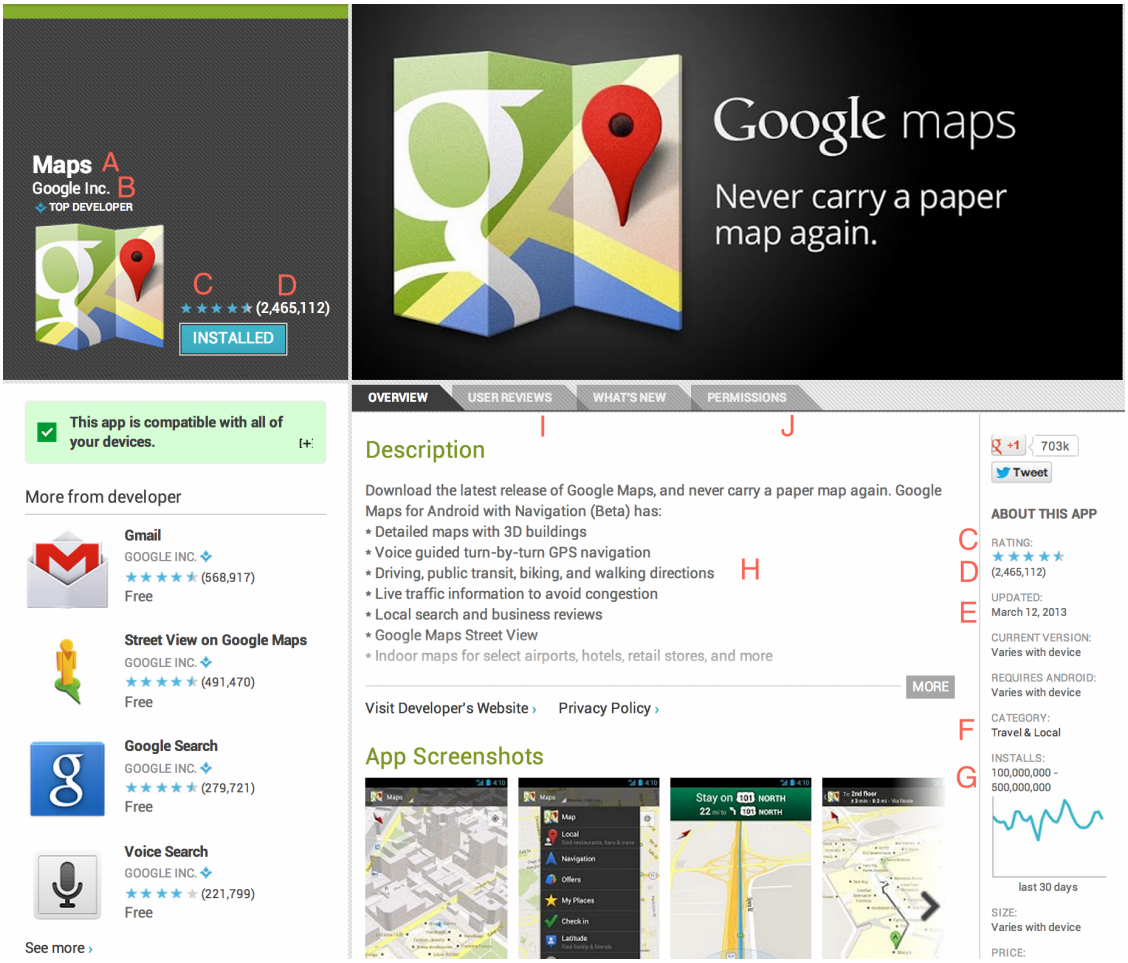

Figure 2.3: A sample page on the Google Play Store, see 2.1

A	App name
B	Developer Name
C	App Rating
D	Number of ratings
E	Date the app was last updated
F	Category in the Google Play Store it falls under
G	Number of installs (range, not exact number)
H	Description of the app
I	Reviews of the app
J	Permissions the app requests

Table 2.1: Various properties of a Google Play Store app page

2.5 Mobile Malware

Malware, as defined by the US Department of Homeland Security, is "Short for malicious software. Programming (code, scripts, active content, and other software) designed to disrupt or deny operation, gather information that leads to loss of privacy or exploitation, gain unauthorized access to system resources, and other abusive behavior" [40]. Like PC OSs, malware is present on mobile OSs, although there are differences.

The tighter security model of Mobile OSs has a notable effect on mobile malware. With tight control in sandboxing, and app distribution, the usual viruses, trojans, and other exploits are more difficult to employ. The main vectors are either OS-level exploits, sneaking past the app review process, or through sideloading of apps. When looking at the two main mobile OSs, a stark contrast is shown. iOS has had "jailbreaking" - privilege escalation exploits - dating back from its first release [18], whereas the first Android exploit was not discussed until 2010 by security researchers Papathanasiou and Percoco [46], and was not seen in the wild until early 2011[15]. On the contrary, no side-loading is possible for iOS, and there have been very few - if any - instances of malware sneaking past Apple's App Store review process, although it has happened[2]. With 95% of all mobile malware[42], Android's malware situation is very much a product of the sideloading and lack of review process found in GPStore[42].

On mobile devices, one of the dominant goals of malware is to gather information that leads to loss of privacy, found in over 28% of mobile malware in 2012 alone[42]. This trend, of malware that possesses no system exploits, but gathers information that leads to loss of privacy, known as Info Theft Malware, is one that Android's Permission-based security model is ill-equipped to handle. Android's permission system relies on the user to determine at install-time if a list of capabilities should be entrusted with the given app. The user is not given a say in how or when the capabilities may be used, nor the ability to reject specific capabilities. At the same time, the mechanisms that keep mobile OSs safe are forcing malware writers to use more subtle techniques, often times without exploits. This all works against the user.

[2]In July 2012, SecureList noticed an iOS app that uploaded all of the user's contacts to a remote location without their consent[37], but others argued this was not as devious as made out to be[57]

In this paper, we attempt to address this key issue through various means. We first introduce several novel concepts for analyzing apps and malware on Android. We then analyze the state of Android apps and Permissions with the most comprehensive android app database available, Android Census. Finally, we propose several novel improvements to the Android security architecture, called AndroMEDA, aimed at building off of our conceptual work.

Chapter 3

Permissions & Security on Android

3.1 Android Architecture Overview

Android is an open source project, built on Linux. Designed to be lightweight, modular, extendable and versatile operating system, Android stripped away almost all of the typical GNU/Linux stack, and wrote an entire framework from scratch. Built in Java, Android runs the Dalivk VM, a lightweight Java-compatible VM (see Figure 3.1).

The three major application components of Android are Activities, Services, and Content Providers. They are all tied together through the Intent system. Activities are user-facing tasks, and follow the iOS definition of an "app". Only one may run at a time, and they have strict lifecycles. Services run in the background, and follow a less strict lifecycle. Their main purpose is to perform long-running tasks that do not require user input. Lastly, Content Providers "manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process" [5].

Android was built from the ground up to be composed of strongly isolated modules with little dependencies. No traditional SysV IPC is allowed, instead Android provides its own inter-app communication built off of its Intent system. Intents on Android, as described in the documentation, are "An intent is an abstract description of an operation to be performed... An Intent provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities. It is basically a passive data structure holding an abstract description of an action to be performed" [6]. Intents allow apps to describe the operation they'd like to perform, without explicitly identifying a recipient. As an example, when the intent *ACTION_VIEW* is sent with data "http://google.com", Android searches through all installed apps that designate that they respond to that intent, and will pick one to deliver it to, in this case, the *Browser* would respond.

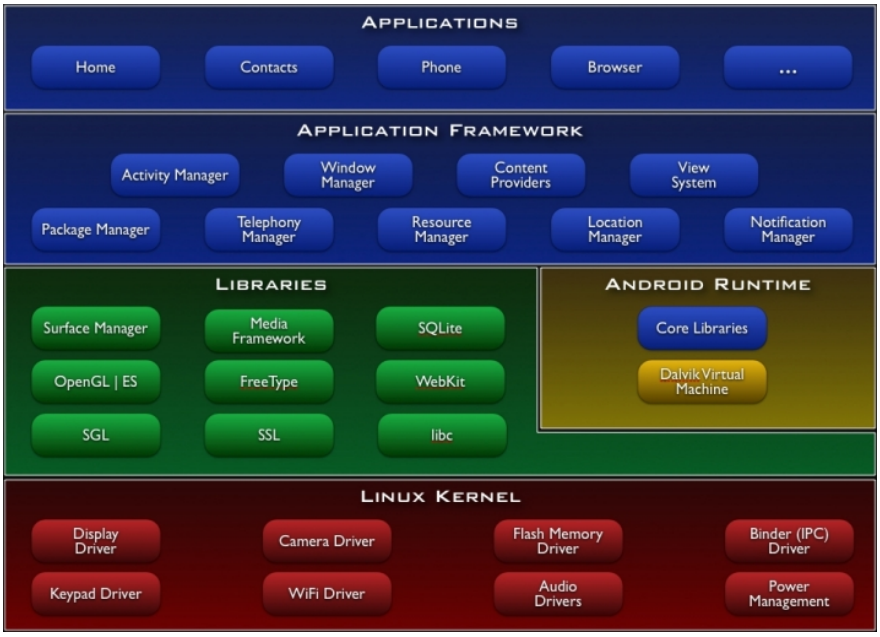

Figure 3.1: An overview of the Android system architecture, from [2]

3.2 Android Permissions

The highly modular and decentralized aspect of Android makes it extremely easy to tap into virtually all Personally Identifiable Information on the device. To protect this, and many other aspects of the system, Android utilizes the Permission security model. The permission security model is a static list of capabilities an app possesses: when presented with this list at install time (see Figure 3.2), a user will either grant the app access to the features, or simply not install the app. When an app requests a permission, the Android system treats it as if the user granted the app that capability. After install, this list will never change unless the app package itself changes, and the user reviews the new permissions.

Android permissions themselves are much more granular than a typical UNIX permission system. They cover a wide variety of operations, from controlling the sleep state, accessing hardware, accessing PII, and many system operations. Some of the most requested permissions can be seen in Table 3.1, the rest can be seen in Appendix Table A.1.

In cases like *WAKE_LOCK* and *CAMERA*, the permission seems fairly singular: access to exactly one feature. However, for other permissions, like *INTERNET* and *READ_PHONE_STATE*, many more granularities could be established, as like *INTERNET* gives unconditional access to all domains, unlike web pages in web browsers. In addition, whereas permissions are intended to be non-overlapping, there still are ways, varying from minor to major, to acquire information guarded by one permission from another. A simple example would be *READ_PHONE_STATE*

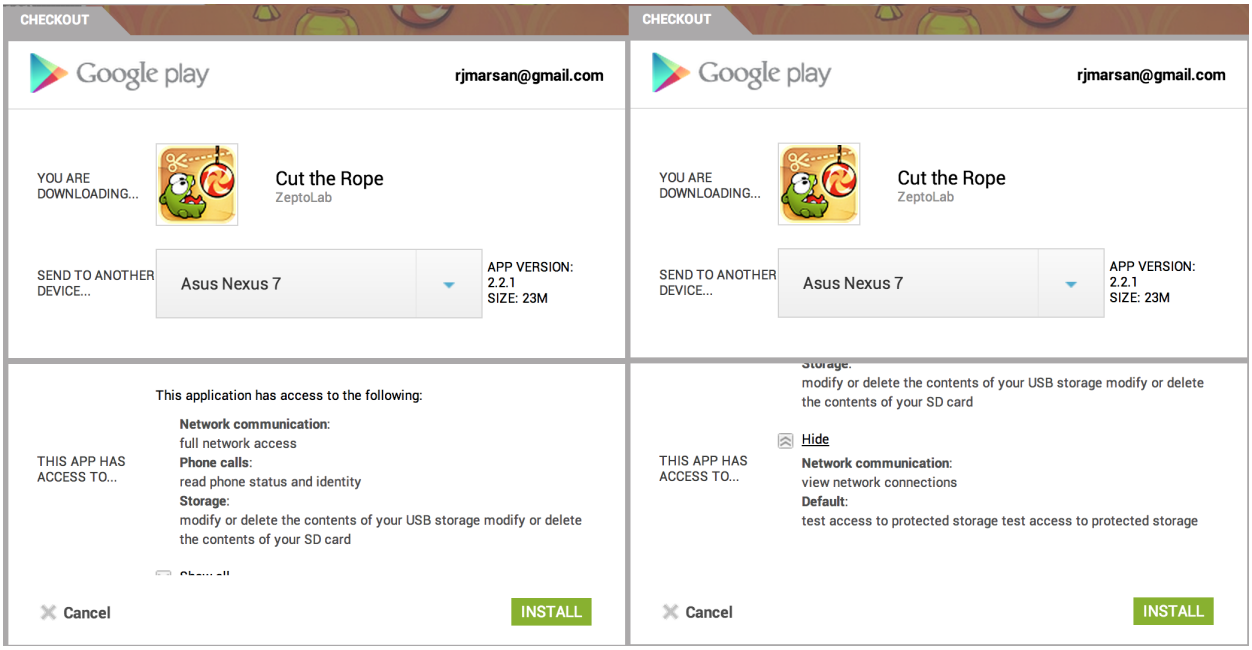

Figure 3.2: A sample Google Play Store install screen showing the permissions. The user must scroll to see all of them, and click "Show All" to see the hidden ones.

would get access to the current phone call, thus being able to establish call logs, something normally protected under *READ_CONTACTS*. However, there are some permissions that are explicitly supersets of another other, like *ACCESS_COURSE_LOCATION* - providing network-tower location, and its superset, *ACCESS_FINE_LOCATION* - providing GPS location.

Permissions do not tend to change through Android's release history. As new hardware is made accessible through Android's SDK, new permissions are added for them, but there are very few times when permissions drastically change meaning or scope. Through Android's 6 year history, only 3 new permissions has been added to restrict previously unrestricted operations[1].

3.3 Permission Enforcement

Permission enforcement on Android is generally performed in two main ways: UNIX permissions and explicit runtime checking. Most hardware is accessible using c or other low-level calls, so permissions are more effectively enforced via UNIX Group Permissions. On app install, Android assigns each app a unique UID, and assigns different group permissions to that user. For example, socket access is granted to a UNIX group, and all apps that request INTERNET

[1] In Android 4.1, the storage device requires a permission to read, and the call logs require a separate permission to read and write - previously they shared a permission with contacts[41]

Permission	Description
INTERNET	Network communication. full Internet access. Allows the app to create network sockets.
WRITE_EXTERNAL_-STORAGE	Storage. modify/delete USB storage contents modify/delete SD card contents. Allows the app to write to the USB storage. Allows the app to write to the SD card.
READ_PHONE_STATE	Phone calls. read phone state and identity. Allows the app to access the phone features of the device. An app with this permission can determine the phone number and serial number of this phone, whether a call is active, the number that call is connected to and the like.
ACCESS_FINE_-LOCATION	Your location. fine (GPS) location. Access fine location sources such as the Global Positioning System on the tablet, where available. Malicious apps may use this to determine where you are, and may consume additional battery power.
ACCESS_COARSE_-LOCATION	Your location. coarse (network-based) location. Access coarse location sources such as the cellular network database to determine an approximate tablet location, where available. Malicious apps may use this to determine approximately where you are.
WAKE_LOCK	System tools. prevent tablet from sleeping prevent phone from sleeping.
READ_CONTACTS	Your personal information. read contact data. Allows the app to read all of the contact (address) data stored on your tablet. Malicious apps may use this to send your data to other people.
CALL_PHONE	Services that cost you money. directly call phone numbers. Allows the app to call phone numbers without your intervention. Malicious apps may cause unexpected calls on your phone bill. Note that this doesn't allow the app to call emergency numbers.
CAMERA	Hardware controls. take pictures and videos. Allows the app to take pictures and videos with the camera. This allows the app at any time to collect images the camera is seeing.
WRITE_CONTACTS	Your personal information. write contact data. Allows the app to modify the contact (address) data stored on your tablet. Malicious apps may use this to erase or modify your contact data.
GET_TASKS	System tools. retrieve running apps. Allows the app to retrieve information about currently and recently running tasks. Malicious apps may discover private information about other apps.
RECORD_AUDIO	Hardware controls. record audio. Allows the app to access the audio record path.
SEND_SMS	Services that cost you money. send SMS messages. Allows the app to send SMS messages. Malicious apps may cost you money by sending messages without your confirmation.
READ_HISTORY_-BOOKMARKS	Your personal information. read Browser's history and bookmarks. Allows the app to read all the URLs that the Browser has visited, and all of the Browser's bookmarks.
READ_CALENDAR	Your personal information. read calendar events plus confidential information. Allows the app to read all calendar events stored on your tablet, including those of friends or coworkers. Malicious apps may extract personal information from these calendars without the owners' knowledge.
WRITE_HISTORY_-BOOKMARKS	Your personal information. write Browser's history and bookmarks. Allows the app to modify the Browser's history or bookmarks stored on your tablet. Malicious apps may use this to erase or modify your Browser's data
RECEIVE_SMS	Your messages. receive SMS. Allows the app to receive and process SMS messages. Malicious apps may monitor your messages or delete them without showing them to you.
WRITE_CALENDAR	Your personal information. add or modify calendar events and send email to guests without owners' knowledge. Allows the app to send event invitations as the calendar owner and add, remove, change events that you can modify on your device, including those of friends or co-workers. Malicious apps may send spam emails that appear to come from calendar owners, modify events without the owners' knowledge, or add fake events.
MOUNT_UNMOUNT_-FILESYSTEMS	System tools. mount and unmount filesystems. Allows the app to mount and unmount filesystems for removable storage.
READ_SMS	Your messages. read SMS or MMS. Allows the app to read SMS messages stored on your tablet or SIM card. Malicious apps may read your confidential messages.
READ_LOGS	Your personal information. read sensitive log data. Allows the app to read from the system's various log files. This allows it to discover general information about what you are doing with the tablet, potentially including personal or private information.
DISABLE_KEYGUARD	System tools. disable keylock. Allows the app to disable the keylock and any associated password security. A legitimate example of this is the phone disabling the keylock when receiving an incoming phone call, then re-enabling the keylock when the call is finished.

Table 3.1: Frequently Requested Android permissions, and the GPStore's description of them

are added to that group. This is simple, effective, and has very little performance overhead. Unfortunately, it makes it difficult to put any additional enforcements as to when and how these resources are accessed.

System operations, like wake locks, changing system settings, turning on and off components, is checked on every API call, through a centralized PackageManager class. The PackageManager first checks if the app has requested that permission. If it has, it then proceeds to check if the permission is protected by the system. Some permissions are only available to trusted code, through either a shared key, or being located in a system folder. Many system operations fall into this category, including *WIPE_DEVICE*, and *BRICK*. These operations typically are extremely dangerous, and have the potential to destroy a device, or perform elaborate phishing operations. Even if an app requests the permission, the PackageManager still has the right to reject the operation. This allows system-level operations to be exposed to trusted developers post-deployment on a device, while still protecting the operations from untrusted developers.

The final aspect of permissions deal with protecting PII. Android implements the bulk of PII sources through Content Providers, which sit in front of a dataset and provide access to remote services. It is therefore imperative for each individual data source to check the permissions of the incoming app, every time it's requested. This, however, provides opportunities to extend on its behavior.

3.3.1 Permission Rejection

When a permission is checked on Android, one of two outcomes is possible. The check passes, and the operation continues as intended, or the check fails, and an exception is thrown. By design, the code path instantly jumps to an error state, halting the action. No logging of this action takes place, nor are partial passes allowed.

3.4 Third Party Permissions

Android was built to distribute PII in a modular fashion, and it extended these features to other 3rd party apps. Any developer may write Content Providers, and likewise, can create Permissions to protect them. Other apps must request that permission before accessing the data. An example of this: Alice writes a messaging app, *MyMessenger*. She wants to expose the PII to other apps, so she makes a Content Provider around it. To protect the user's information, she defines her own permissions: *mypackage.MyMessenger.READ_MESSAGES* and *mypackage.MyMessenger.WRITE_MESSAGES*. Bob writes an app that uses *MyMessenger*'s data to build a picture. His app requests the permission *mypackage.MyMessenger.READ_MESSAGES*. When his app contacts Alice's Content Provider, she checks this custom permission before proceeding with the query. However, if Bob does not request

mypackage.MyMessenger.WRITE_MESSAGES and attempts to perform a write action, Alice's Content Provider will reject the operation.

If two apps request the same permissions, as long as neither have access to system permissions, it can be assumed that they have the same capabilities. These capabilities are permanent as well, as permissions for a given package can not be added nor removed. This Permission Fingerprint, or set of capabilities, uniquely defines what an app has access to. All apps that request the same set of permissions have access to the exact same set of actions, and only through those permissions do they have those actions (with a few exceptions). However, Android permissions are static, therefore a Permission Fingerprint doesn't guarantee a specific pattern of behavior. In fact, since not all permissions are guaranteed to be granted, a Permission Fingerprint simply establishes the absolute maximum capabilities of an app - even if the system rejects some of them.

Chapter 4

Malware on Android

Malware on mobile devices has seen a departure from past exploits. The wealth of Personally Identifiable Information easily available on mobile OSs has made them the increased focus of malicious software. In addition, the tight sandboxing constraints have often forced malware writers to either find exploits to break out of the sandbox, or to write malware that cloaks itself as benign. Without finding exploits, code may not be ran by the user unless it is in *app* form, and come through a trusted channel (unless that security feature has been disabled). Once on the device, malware has several main methods of attack.

4.1 Installation

The three primary ways that malware gets installed on Android are *repackaging*, *update attack*, and *drive-by download*[67]. The first two are designed to sneak malware into the Google Play Store or other 3rd party stores, and the third is designed to trick the user into installing it by mistake. *Repackaging* deals with the technique of taking an existing app, adding malicious code to it, and repackaging it again, discussed further in Chapter 8.2. *Update Attacks* typically build off of *repackaging*, but do not acquire malicious code until later - making static detection difficult[67]. The last, *Drive-By Downloads*, is often tied with *repackaging*, but is not presented in an official app channel. Rather it is downloaded when the user visits a webpage, or clicks a link [67], and Android prompts the user as to whether they would like to install the app. All three methods involve concealing the intent of the malware, and passing off as a legitimate app, they do not use browser exploits or system exploits to install the initial malware without the user's consent.

4.2 Malicious Actions

Once installed on the device, Android malware has several main categories of attack. Xuxian Jiang and Yajin Zhou[67], along with Spreitzenbarth[54], and Hackmageddon[28] define 4 major categories: *Privilege Escalation Attack*, *Remote Control Attack*, *Monetary Service Attacks*, and *Privacy Info Theft*.

4.3 Privilege Escalation Attack

Privilege escalation attacks take several forms on android. The basic premise is simple: acquire access to operations beyond what the sandbox and granted capabilities provide. The main way to accomplish this is via *rooting*.

4.3.1 Rooting

Rooting is the act of acquiring root, or administrative privileges, on an OS. Typically mobile OSs do not provide the user, or apps, with root capabilities, and instead reserve that for a set of trusted system processes. However, by finding vulnerabilities in these services, or exploiting the OS itself, apps can escape the sandbox. After an app has been granted root capabilities, the permission system no longer applies to it: it can simply access whatever it wants. These attacks are difficult for the system to detect: all monitoring of apps relies on monitoring the sandbox - when an app escapes that, there's no tracking it. This technique is commonly employed by botnets - giving remote access to the core system.

Many examples of root exploits exist, dating back to 2011 with *RageAgainstTheCage*[35]. These root exploits were very popular for non-malicious purposes, circumventing the device's sandbox to install a permanent root binary, creating a similar setup to a typical UNIX computer, where root may be acquired after a password/permission. However, in March 2011, DroidDream was discovered. DroidDream used *RageAgainstTheCage* to silently install additional applications in the background. From there, it proceeded to steal PII and become a botnet. By the time it was remotely removed from the market, it had been downloaded an estimated 50,000 to 200,000 times[15] - the largest bulk-remote-removal of apps seen from the GPStore. From then on, a stream of root exploit malware was found, based off of *RageAgainstTheCage*, *udev*, and one exploit called *GingerMaster*[26].

Recent Android Rootkits

GingerMaster[52] is significant because it is the last known root exploit malware seen in the wild. Designed for Android 2.3.3, it can currently run on 45% of all Android devices in use, according to the official Android Dashboard[4]. However, in reality, of that 45%, almost all of them run Android 2.3.3 or higher, and virtually all have been patched to fix the exploit[52]. All previous exploits were patched in Android 2.3[52], meaning less than 6% of all active Android devices are vulnerable to them. Since Android 4.0, Google has focused greatly on security, improving ASLR[24] and hardening system services[7]. No known rootkits exist - malicious or not - for Android 4.0 and up - 54% of active Android devices.

4.3.2 Confused Deputy Attack

The second main vector for privilege escalation attacks is the Confused Deputy attack[29]. In this scenario, services that guard sensitive operations are "tricked" into performing them. An example of this would be if a Content Provider forgot to check a permission, or finding APIs that do not correctly perform a permission check. Perhaps the simplest example of this is the ability for any Android app to contact remote servers, simply by asking the web browser to open a URL. By passing sensitive data in the URL, an app may still contact a remote server without ever requesting the *INTERNET* permission. Projects like XManDroid[13] and Quire[19] address this by extending Android to analyze inter-app communication and detect this kind of attack.

4.4 Remote Control Attack

Remote Control Attacks, frequently called *botnets*, are the ability for malware to accept commands from a remote server, controlling the device. This technique is common, Xuxian Jiang and Yajin Zhou[67] found in 93% of Android malware, and is often used in conjunction with other attacks[54].

4.5 Monetary Service Attack

The second malware technique is possibly the simplest: perform services on behalf of the user that cost money. Examples of this include calling costly phone numbers and sending premium SMS messages. Typically these actions are performed without notifying the user, and only visible after the user checks their bill. These attacks have been prevalent in the Android market for quite a while, with NQ Mobile[42] listing it as one of the top 3 threats of 2012, and being found in 39 of 119 of the malware documented by Spreitzenbarth[54]. However, recent versions of android - after 4.2 Jelly Bean[7] - have taken the step of warning the user before premium SMSs are sent.

A prime example of this is FakeInst[60], a repackaged version of Instagram[32] that sent premium SMS messages on start. "In the background, the fake downloader sends a premium rate SMS to the number based on the country of origin for the user"[60]. In many cases, the premium SMS messages would end up being billed to the user for over $4 each[60], without ever alerting the user. Messages are often deleted by the app, removing the trace until the user gets their bill.

4.6 Private Info Theft

The last malware technique is the most significant, and represents the largest departure from typical malware. Apps that steal PII, or Info Theft Malware. The theme is fairly straightforward: Provide the user with a seemingly legitimate

app, but in the background acquire large amounts of valuable data, from call logs to contacts to photos, and send them to a remote server. This fits in well with the main themes of mobile computing: the consolidation of many sources of PII all in one device. To the system, no unusual operations are performed, and no exploits are ran. The qualification for Info Theft Malware lies in the "use" vs "misuse" of PII, often times, this line is blurred.

4.6.1 Path on iOS

A large distinction in what constitutes as privacy malware to an individual stems from their expectations of how the app will use their PII. Consider the case of the Path iPhone app, which in Feb 2012 was discovered to be uploading the user's entire contact list to Path's servers, without any permission from the user[55]. It's fairly uncontroversial for a social network to read your contact data, and the act of scanning contacts to help "find your friends" on Path wasn't out of the ordinary. As VentureBeat discovered: "Facebook, Twitter, Instagram, Foursquare, Foodspotting, Yelp, and Gowalla are among a smattering of iOS applications that have been sending the actual names, email addresses and/or phone numbers from your device's internal address book to their servers"[58]. Ultimately, however, the outrage was sparked because of how unexpected the behavior was.

The Path incident sparked several key changes in iOS's security model: specifically, having a popup occur when an app requests access to the contacts database, and allowing the user to reject the request. This, in general, is a one-time request, after which the app is given free reign over content[1], which doesn't fully address situations like Path, where it's less about the app simply having access to the data, but what the app actually did with the data behind the scenes. When these actions did not match up with user expectations, it was treated as malware until the situation was cleared up by Path. The next day, they issued an update immediately explaining to the user what they were going to do with the data, and why.

It's worth noting that the only reason Path's contacts upload mechanism was discovered was by accident: Arun Thampi discovered it as part of a company hackathon, and only via sniffing the HTTP requests coming from the phone. An ordinary smartphone user would not have access to these tools, nor have the time and patience to sift through the data to spot unusual behavior. These actions are unchecked and hidden from the user, not giving them a chance to decide for themselves if they are comfortable with them - supporting our motivation for AndroMEDA.

4.7 User-App Agreement

This incident, however, lies at the heart of mobile malware: Misuse of PII lies in the abstract definition of how the app is expected to behave. Apps that violate this expectation of behavior are classified as malware, and apps that do not are not. This agreement between the user and the app, the User-App Agreement (or UAA), is an informal understanding

the user has as to what actions an app will take. This differs from the Permission Fingerprint, which is a measure of what actions the app is capable of performing, instead dealing with exactly when and how those actions are taken. Since this agreement is not formally defined, it's acquired through external trust in an app. This happens in various ways, through the description the app provides, to the knowledge and referral of the app from other trusted sources, or the trust in the developer. The UAA is not a measure of how trustworthy an app is, but rather a framework for the user consenting and trusting specific actions an app may take.

4.7.1 UAA Example - Social Networking App

An example of UAA can be seen in the expected behavior of a hypothetical app and user. The first is a large Social Networking app, which requests permission to access internet, send SMS messages and read the contacts database. If the Social Networking app accesses contacts when the user requests it "find my friends", and it sends SMS messages after the user messages another user who is not "online", these actions fall within the UAA of the Social Networking app and the user. However, if the contacts database is read and uploaded to a remote server without the consent of the user, this may violate the UAA, breaking the trust of the user (building off the example of Path).

4.7.2 UAA Example - Social Game

The other example is a little known developer's game, which also requests permission to access internet, send SMS messages and read contacts, same as the Social Networking app. These documented capabilities in the Permission Fingerprint may be enough to violate the UAA: the user may not trust an app with the capability of these actions. However, in the case that the user does, or simply doesn't pay attention, the app still may not violate the UAA. If the app is an online game, and asks you to find other people you know who are playing it, this would more likely than not violate the UAA. However, if it sends SMS messages to your friends telling them to download the game, this would breech the UAA, breaking the trust of the user.

In both examples, the apps have the exact same Permission Fingerprint, but vary wildly in their expected behavior, and which actions are trusted and untrusted. This fits right along with our definition of malware, with the misuse of PII and other device capabilities. This also highlights the shortcomings of the Permissions framework - being unable to deal with the subtle differences between trusted behavior and untrusted behavior. Indeed, for any given user, they may have a very different understanding of what acceptable behavior is. Therefore, UAA plays a crucial role in classifying apps in relation to Info Theft Malware.

4.8 Proof of Concept Malware in Academia

In the realm of malware research in academia, several prominent proof-of-concept examples further demonstrate the vague line between use and misuse of PII, and our concept, the User-App Agreement. The most notable one is SoundComber[50] It passes off as a benign app, but in the background records audio, and does on-phone processing to find sensitive PII, after which it uploads the information to a remote server. This app is unique because of its simple Permission Fingerprint, and its ability to gather sensitive PII from a channel not suspected to be very rich in PII.

The second prominent example of academic malware on Android is TapLogger[64]. TapLogger imitates a simple touch-based game, learning the vibration patterns of the device for each tap. After which, TapLogger records the vibration patterns in the background, attempting to discover passwords and other sensitive keyboard events, all through a seemingly trusted sensor. TapLogger requests no Permissions, therefore its behavior is a possible behavior of virtually all apps.

In Chapter 8.2, we build upon these examples to present an additional dataset of research IncognitoWare, or repackaged apps with malicious software added. We keep the Permission Fingerprints identical to the cloned app, making detection exceptionally difficult.

4.9 Conclusion

The landscape of malware on android follows many clear patterns. The first is the use of masquerading as benign apps, and passing through trusted/semi-trusted channels to enter the device. Once on the device, the four main categories of attacks are privilege escalation attacks, remote control attacks, monetary service attacks, and info theft. Of these three attacks, privilege escalation and monetary service attacks are the easiest to protect against, and indeed Android has taken serious steps to mitigate these. However, the third time of attack, Info theft, is the most difficult to mitigate on Android, due to the shortcomings of the Permissions framework, and the wide spectrum of severity these attacks can take. Since these attacks may vary in interpretation per user, and lay in the subtle communication between the user and the app, we highlight the need for a concise representation UAA, where the user can evaluate the actions themselves.

Chapter 5

Related Works

5.1 Android Extensions

AndroMEDA is far from the first to attempt to address the issue of malware on Android. As early as 2009, frameworks like SAINT[43] built off of the Android Permission Framework by implementing runtime policies the user could define per-app. Later on, projects like TISSA[68] built off of this by implementing varying levels of obfuscated data. When an app running on TISSA android requested access to the contacts database, TISSA could either provide the app the full database, some limited portion, some anonymized portion, or outright returning no information. Not all PII was covered in this, nor were any temporal rules established: The system behavior for a specific database of PII for an app was consistent across requests. These events were not shown to the user either, something addressed by TaintDroid[21].

TaintDroid is a novel extension for Android, focused on flow detection of PII. By modifying the low level VM of android, Dalvik, variables are tainted once they access PII. This taint flows throughout the system, and when the variable reaches a designated exit location, the event is logged and alerted to the user. Finally, YAASE[49] is a relatively new security extension, but combines many aspects of TaintDroid and TISSA, to become an extremely powerful way of detecting information flow and prevention.

There are several main shortcomings in all of these frameworks. First off, they all require significant modification of the Android codebase, thus creating very difficult work for developers who seek to incorporate these extensions into their Android OS. Performance impact, although somewhat negligible by many accounts, certainly plays a role in a decision to incorporate such security extensions.

Another important shortcoming is the complex rulesets required. Most build off of the Permission Framework, adding additional rules the user may configure. However, a study by Berkeley suggests only 17% paid attention to permissions at install time, and only 3% correctly remembered them later[23]. Clearly, tasking the user with more work is not the right approach. Looking at malware through the context of the UAA, all but TaintDroid fall short in one key regard: alerting the user of suspicious behavior.

Several frameworks have touched upon the concept of the UAA. Andromaly[53], developed in 2010, pBMDS[63] and Crowdroid[14] all attempt to classify malware based upon its interaction with the user. However, none actually ask input from the user - a fundamental flaw that limits the ability to adapt to the user's specific UAAs.

5.2 Android Sandboxes

On the other side of malware detection is automated malware detection. The major project in this regard is Google's Bouncer[34]. Introduced in 2011, Google Bouncer is a system that runs malware in a highly-observed sandbox, and watches for suspicious behavior. Since its release, it's been the subject to quite a bit of criticism[36], with researchers finding over 20 ways to circumvent it.

Along similar lines, TrendMicro provides its solution, App Reputation[56]. It runs apps in a "cloud" sandbox, watching for connections to suspicious websites, as well as other monitoring. A research project, Paranoid Android[47], runs in a similar vein, monitoring apps in a sandboxed Android OS. However, in order to get accurate information on app behavior, actions must be recorded.

However, most all of the frameworks listed above, especially the sandbox tools, assume a clear ruleset to be established classifying malicious behavior vs benign behavior. They require a clear line to be established as to what constitutes trusted behavior vs untrusted. Unfortunately, this is counter to the concept of the UAA, where every individual action has a complex set of rules that result in an acceptable behavior vs unacceptable. Users may also have vastly different rules for what constitutes malicious behavior. Apps that send the Unique Device ID (UDID) and location information to ad networks might be malicious to some users, and perfectly normal to others.

5.3 Conclusion

The main counter argument to all of these frameworks is the example of SoundComber: It records in the background, looking for PII. Some frameworks and policies may immediately flag this as malware, but one can conceive of perfectly benign apps that would follow this exact same formula: Dictation apps that transcribe speech to text over long portions of time, or audio broadcasting utilities. Apps that take all PII and upload it to a server may be classic malware, or simply a backup utility. Context and Use, and more generally the UAA, is an extremely important part of malware detection that's missing from modern android security frameworks.

Chapter 6

Android Malware Evaluation Detection and Analysis

6.1 Goals

Android Malware Evaluation Detection and Analysis (AndroMEDA) is designed to complement existing Android security extensions like TISSA[68], TaintDroid[21], YAASE[49] and XManDroid[13]. The goal in building AndroMEDA was the minimal amount of changes to the Android system, to extract sufficient amounts of information. Portability was important, as code that is easier to port can be more easily adopted. Along with the goal of being lightweight and fast, this further forced the extension to be as simple as possible. The framework aimed to be as independent of hardware as possible, as to work on any device - smartphone, tablet, TV, or future devices like Google Glass. This meant as little changes to the low-level drivers and kernel as possible.

Functionally speaking, the main goals of the framework are to better understand how apps behave, what PII and capabilities they access, and to provide the user with the information necessary to quickly evaluate an app's actions in relation to its UAA. Logging sensitive events, and finding strategic opportunities to expose them to the user, giving the user actions to perform in reaction, is a key step in mitigating Malware, especially Info Theft Malware, on Android. Unfortunately, as discussed in Chapter 3.3.1, Android does not log these sensitive events natively, motivating this framework. Ultimately, it's expected that a framework like this will not eliminate malware on android, but rather be part of a larger holistic system.

As a base for our framework, we chose CyanogenMod[16]. CyanogenMod is a 3rd party open source Android distribution created by volunteers. It provides firmware for a wide variety of Android devices, and its user-base are people seeking to replace their stock OS. This makes it an ideal candidate to fork our framework from - our changes can therefore be incorporated into the over 4.1m+ installs of CyanogenMod[17].

6.2 AndroMEDA Architecture

Android, as an open source project, is hosted at `source.android.com`[8], as a series of *git* repositories managed by a meta-script called *repo*. Its source tree is organized by project type, "frameworks", "external" and "libcore"

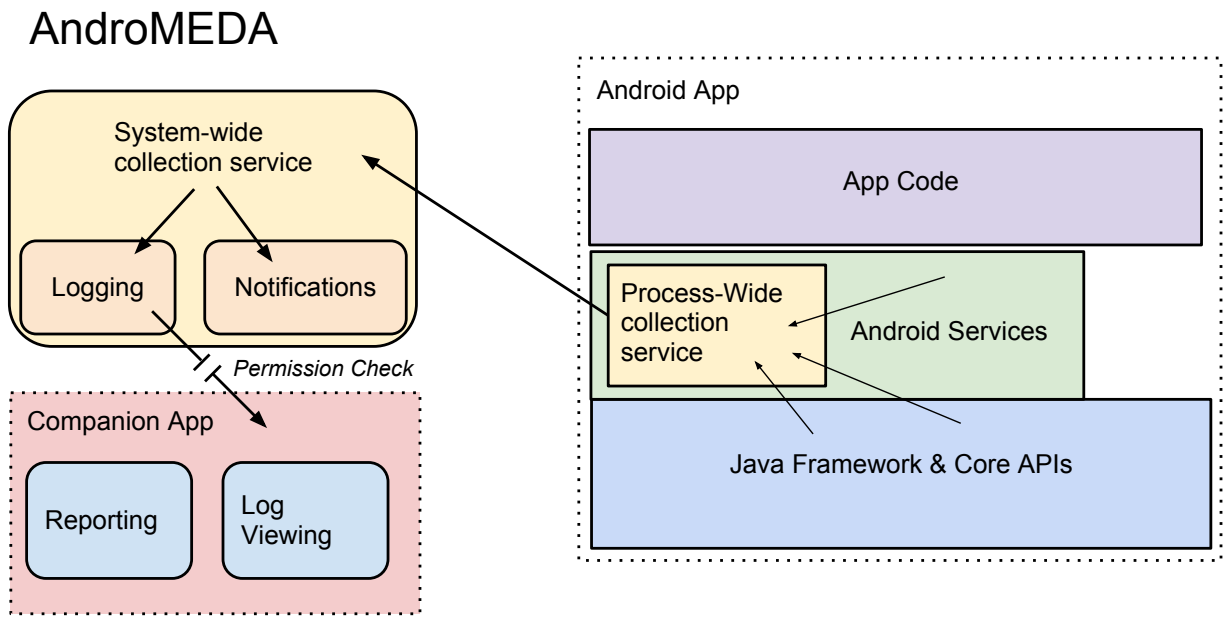

Figure 6.1: AndroMEDA Architecture Overview

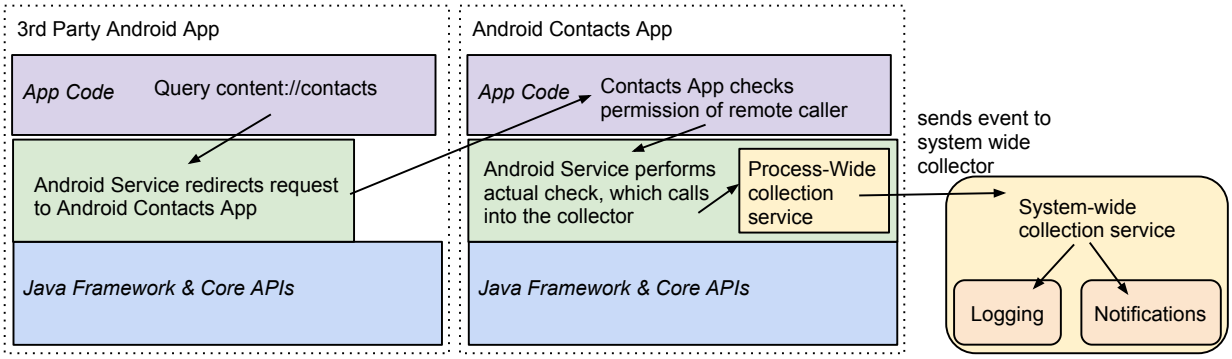

Figure 6.2: Example code path of an app requesting data from the Contacts app

being the top level folders we focus on. “frameworks” contains all of the java, c++, c, assembly, and other code that compose the core framework that runs on top of the main libraries. “libcore” and parts of “external” comprise the main libraries. The bulk of the code lies in “frameworks/base”.

The architecture, seen in Figure 6.1, is organized into two main parts: A collection of hooks in the API, and a system service to collect this information. The collection of hooks in the API calls into a process-wide service that translates them into events that get sent to the global system service. At the start of every new Dalvik process, the process-wide collector installs hooks into the framework that notifies the collector when the APIs are called.

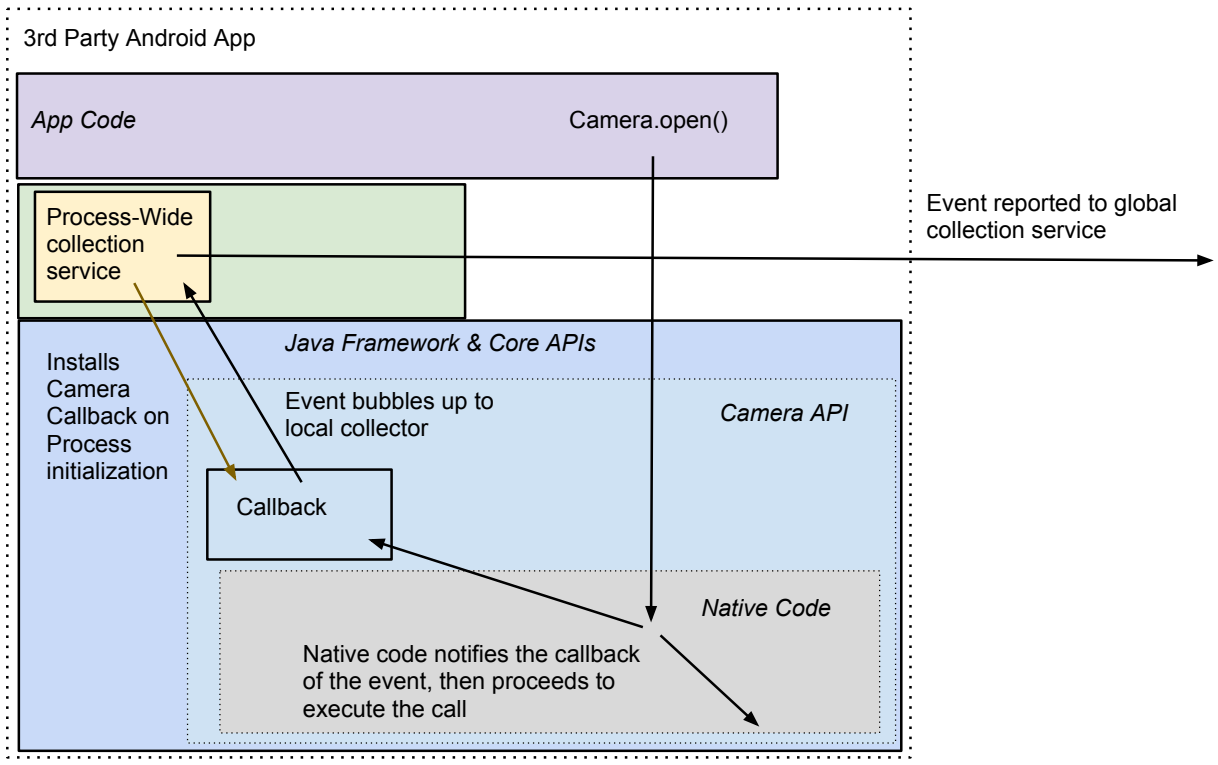

Figure 6.3: Example code path of instrumenting the Camera API

The hooking mechanisms vary depending on the exact call being instrumented. Android has a standard mechanism for checking permissions in many cases, and we placed a hook where this is enforced. Usually, this takes place during an Remote Procedure Call (RPC) with the process that owns the data - and such, the remote UID is sent along, as seen in Figure 6.2. Unfortunately many permissions are not checked through this manner, but Android Permissions Demystified[22] has a comprehensive list of API calls and the permissions they require, making it easy to locate essential APIs to instrument. For the more difficult APIs, a static global callback variable is placed in every class we want to instrument. Upon launch of the process, our local collector populates these global variables with callbacks that marshal the data off to the main collection service. Calls are then made to the callback at the appropriate times in the API's normal function, seen in Figure 6.3. Using this method, we are able to instrument any API call, getting more data than simply when the permission is checked.

For c and c++ level code, we establish hooks in the Java Native Interface (JNI) - Java's method of communicating with native code - to call back to the main object, since using JNI to call back into Java from c is cumbersome and error prone, which then calls the process-wide callback accordingly, an example can be seen in Figure 6.3.

The ability to log virtually any API call is extremely valuable. Not only are permission events logged, but more fine grained events like the starting and stopping of an audio recording can be measured. Internet sockets are logged,

along with all POST data set out. Not only is this system light-weight and fast, but it does so with as little modifications as possible. No kernel-level or dalvik-vm-level changes were required.

The global service serves as the central point of collection of all events. It runs in its own process, and uses Binders (Android's RPC) to communicate with other processes, which feed events into it. These calls are asynchronous, and do not block the remote caller. The service is responsible for logging and processing the events, and taking action on them.

Two way communication is also possible. An app's local collector can query the global service and acquire a list of events for a given package, among other things. This allows user-level apps to be written to leverage AndroMEDAs findings.

6.3 Companion App

A companion app was written, to allow the user to inspect the information being gathered by AndroMEDA. The companion app responds to the Intents broadcast by the notifications that AndroMEDA displays to the user. This brings up a history of events logged by that app, and allows the user to take action. The user may report an app, or uninstall it. A simple web service was set up to aggregate these reports.

The companion app is significant because it does not require the AndroMEDA framework to be present on the phone to function. Despite no local logs being available, the companion app can still query the web service, and view event histories that other users have published.

Chapter 7

Market Analysis

In order to evaluate AndroMEDA effectiveness in identifying malware, it's important to evaluate the ability to identify malware without it. As previously discussed in Chapter 3, Permissions are the single security measure that defends an Android user from malicious software. We show a fundamental disconnect between Permissions and UAA, as conclusive evidence that Permissions should not be the sole security system on Android. In order to study this, we must examine the how Permissions reflect app behavior, and if known malware can be identified with Permissions alone.

7.1 Android Census

To do a broad scale study of Android permissions, we introduce a novel dataset: Android Census[3]. Android Census contains a rich dataset of metadata of apps in the Google Play Store. We first obtain a list of all packages in the Google Play Store from AndroidPit[9], a 3rd party source that maintains an up-to-date index. We then crawl the GPStore using this list, and insert the data into a MySQL database. See Table 7.1 for statistics. The metadata described in 7.2 is rich in contextual information about an app. The GPStore was scanned twice per day, generating over 22 million entries, for a period of 1 month in May-June 2012, after which it was stopped[1]. This dataset provides detailed metadata for all apps included in our package list, including the most common apps, but also a larger dataset of lesser known apps (see Figure 7.5). Despite the large amount of metadata available, we will focus on the Permissions, installs, and category, leaving most other fields for Future Work (see Chapter 9.2).

Scan Timestamp	Number of Apps
fill this in	I should fill this in

Table 7.1: Statistics from Android Census

[1] We plan on continuing scans in the future

Metadata	Description
App Name	The name of the app, e.g. Google Maps
App Developer	The name of the developer, e.g. Google
Android Version	The lowest compatible Android version
Number of Installs	The total number of installs. Given as a range.
Description	A long (3000 word max) description of the app.
Reviews	The user reviews of the app.
Overall Rating	The overall rating of the app, from 1 to 5. A user does not need to write a review to leave a rating.
Requested Permissions	The list of all the permissions the app requests.

Table 7.2: Metadata from Android Census

7.2 Global Permission Analysis

We first look at the global trends of Permissions in the GPStore. Ideally, we are in search of a system where access to sensitive PII and dangerous operations is only requested by a handful of apps that truly need them. This would both increases the user's ability to understand how an app behaves, increases the user's trust in the Permission system, and demonstrates a strong connection between Permission Fingerprint and User-App Agreement. Ultimately, however, we find a system that falls short of these goals.

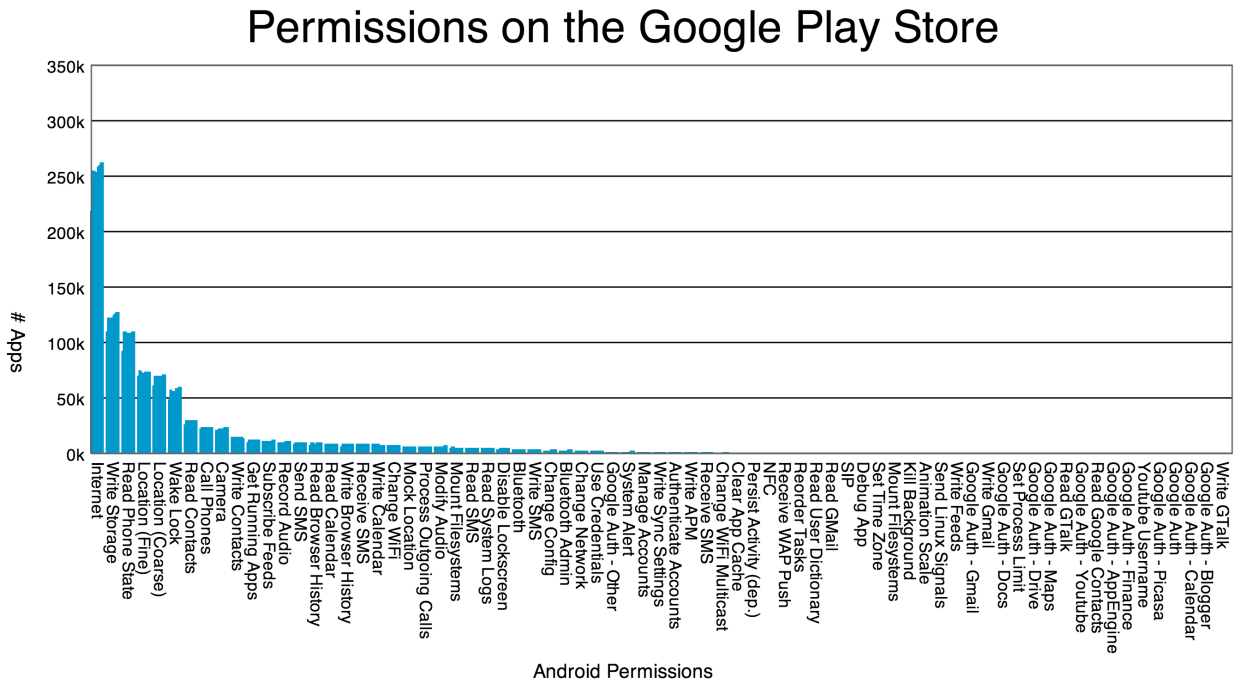

Figure 7.1: Permissions, sorted by how many apps request them in the entire GPStore dataset

Figure 7.1 shows all major permissions in the GPStore, sorted by frequency of use. This graph highlights several things: First off, *INTERNET* is a dominant permission, with well over half of the GPStore apps requesting it. The next 5 permissions, *Write to Storage*, *Reading Phone Info*, *Location Info* and *Wake Lock*, all have over 50,000 apps

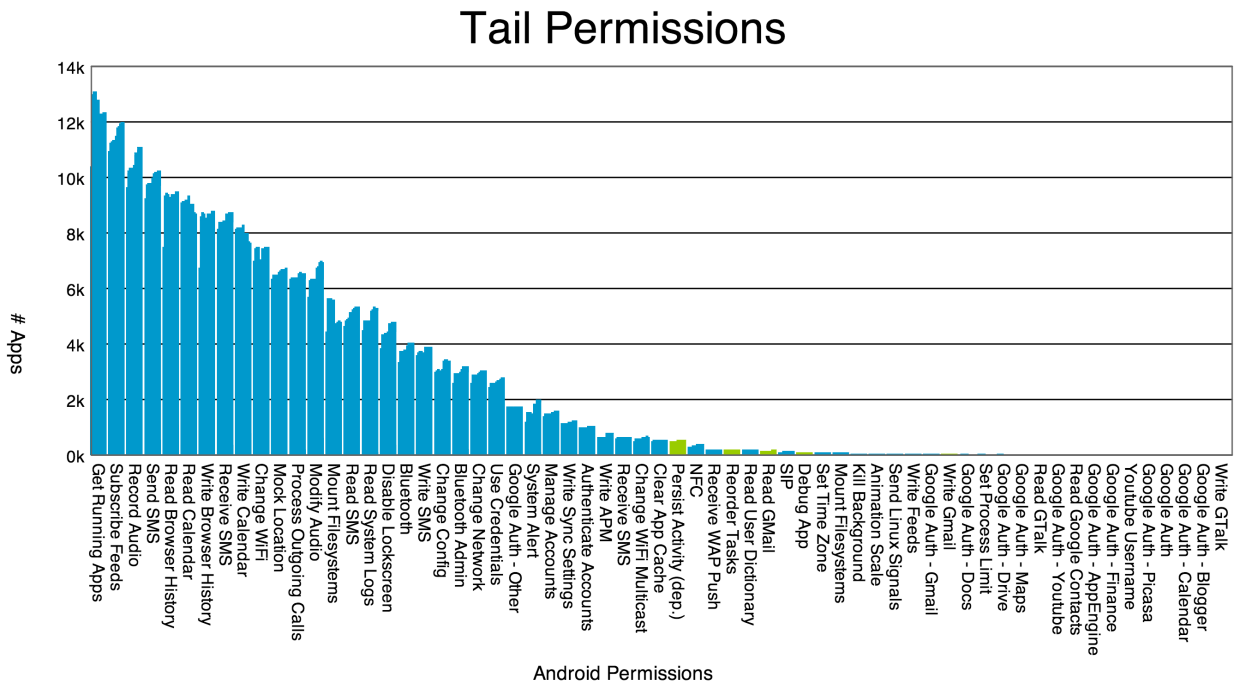

Figure 7.2: Less commonly used permissions in the GPStore

that request them. A steep drop is seen for *Read Contacts*, *Call Phones*, and *Camera*, with around 25,000 apps each. Of these top 9, 3 provide access to semi-personal information: *Reading Phone Info* and the *Location Info* permissions. It's only when we get to the lower 3 do we get access to personal information.

Figure 7.2 examines the tail permissions in the GPStore. This section contains the bulk of permissions related to PII, with *Record Audio*, *Read Calendar*, *Read SMS*, and so on. It additionally contains many sensitive operations, like *Send SMS* and *System Alert* - which allows apps to draw windows over other apps. These permissions, while being a relatively small portion of the GPStore, still occupy a substantial portion. Indeed, with every dangerous permission, many legitimate use cases can be established that would not violate the UAA, see Figure 7.3 for examples.

7.2.1 By Market Category

Overall, we find many PII related permissions are requested a substantial number of times. A number of use cases exist, but it remains to be seen if the apps follow those use cases. To further examine this, we separate the apps in Android Census into the categories present on the GPStore, and rerun the analysis, seen in Figure 7.3.

Immediately, the use cases described in 7.3 become apparent. A large spike is seen in category *Photography* and *Media & Video* for permission *Camera*, similarly with *Record Audio*, although it's less visible in *Music & Audio*. Location data is commonly used in *Weather*, *Transportation* and *Travel*, another use case. *Send SMS* is found predom-

Permission	Use Case
Location (Fine)	This one has a wide variety of uses, from location-specific news apps and games, to social networks. It's also used by ad networks included in many apps.
Location (Coarse)	This one follows the same trends as *Location (Fine)*, but is skewed towards ad networks.
Read Contacts	Any access to the address book at all require this, so communication apps, social networks, or many apps that involve sharing with friends will use this.
Call Phones	This one is oddly popular. Many customization apps, especially those that seek to replace stock Android apps, will use this, especially if they are replacing address book or home screen functionalities. Additionally, many communication apps will use this, for obvious reasons.
Camera	Any photography app or video camera app will make heavy use of this permission. This functionality is often present in other apps as well, e.g. taking a photo of the user for use as a profile photo
Write Contacts	This permission is often used with *Read Contacts*. Many social networking sites and services wish to provide "contact syncing" abilities with Android device, which requires having write-access to the Contacts database.
Record Audio	Like *Camera*, audio apps and communication apps make heavy use of this.
Send SMS	Many apps seek to replace the default SMS app, which would therefore require all of the SMS related permissions.
System Alert	This permission protects drawing on the screen, on top of other apps. Many apps are designed to be on the screen at all times, either replacing Android components, or complementing them.

Table 7.3: Use cases for common Android Permissions

inantly in *Communication* and *Social*, another good sign. *Wake Lock* was found heavily in games, music and media, and communication, but less so in other categories, as would be expected for Its use case.

Ultimately, however, some odd patterns can be observed. Despite *Read Contacts* being popular for apps in *Communication*, *Business* and *Social*, it's also found heavily in the *Brain & Puzzle* game category. A similar pattern is seen in *Call Phones*; Its predominant category is *Communication*, but it's found in significant amounts in *Medical*, *Shopping*, and *Lifestyle*. We also find a spike in *Get Running Apps* in games that does not appear to be present in other market categories. Perhaps the oddest observation was the extremely large spike of *Read Browser History* and *Write Browser History* found in the *Personalization* category.

Isolating the apps in the dataset from category *Personalization* that use the permission *Read Browser History* provides insight into this odd pattern. The most popular app in *Personalization* to use the permission is an app called Launcher Pro[33]. Downloaded over 10 Million times, with an average rating of 4.5, LauncherPro is a highly respected 3rd party Android Home Screen. It has 181,628 5 star ratings, and only 8,230 1 star reviews. The permission comes from a set of widgets to access the browser shortcuts from the home screen.

The next most popular apps in *Personalization* that use permission *Read Browser History* are Seabed Live Wallpaper[51] and Heart Live Wallpaper[30], both by the same developer "Good LiveWallpaper"[27]. These two apps have each been downloaded over 1 Million times, and have an average rating of 3.8 and 4.1, respectively. However, the bad reviews for both apps make numerous references to poor behavior, with one user saying "The sheer number of push ads this single dud of a wallpaper shoves on your phone is astounding."[51]. It's clear these apps does not use *Read Browser History* for a trusted purpose, but rather for advertising.

Despite the fact that the two apps come from the same market category, and use the same permissions, they have vastly different expected behavior and UAA, one of which violates it frequently, and the other is highly trusted. Permissions, while being used rather judiciously in many parts of the market, clearly have times when they are requested without truly needing them. These situations degrade the effectiveness of permissions, because the user lacks a trustworthy reason why the app requires such capabilities.

7.3 Android Malware Genome Project

Next, we project a malware dataset onto the Android Census, with the goal of identifying their Permission Fingerprints as potential outliers. Ideally, all malicious behavior would show up in the permission fingerprint of an app, so therefore Its unique set of capabilities would stand out in the Google Play Store, thus showing a strong correlation between expected behavior and Permission Fingerprint. Upon further exploration, we find many patterns of malware are easily identifiable through the Permission Fingerprint, but some behavior passes by unnoticed, demonstrating the need for AndroMEDA.

The dataset we use - the largest academic set of Its kind, is the Android Malware Genome Project[67], from NCSU. Containing almost 1300 samples from 52 families of actual Android malware, it provides the ideal test dataset for Android.

Malware Class	Description
Root	The malware uses a rootkit as a method of attack.
Botnet	The malware exhibits behavior associated with botnets, e.g. accepts remote commands.
Banking	The malware is designed specifically to intercept Banking messages.
SMS	The malware sends Premium SMS messages charged against the user.
Info	The malware uploads personal information to a remote server, without notifying the user
Spyware	The malware remains on in the background, or has the capability of remotely monitoring the smartphone user.
Market	The malware was spotted in the official Google Play Store

Table 7.4: Malware Classes found in the Android Malware Genome Project

7.3.1 Classification

We first classify each malware family by its capabilities, according to data aggregated by Spreitzenbarth[54] and Hackmageddon[28], with the chart shown in Figure 7.4, and the explanation of the classification of capabilities in Table 7.4.

Of the three classes of malware discussed in Chapter 4, we observe that very few of the malware families use only one attack vector - 6 families, all of which only used *SMS* attacks, and 1, *Pjapps* used none, only being a botnet.

Clearly, Android Malware often uses multiple vectors. We also note that 33 of 52 - 63% of all malware families upload personal information to a remote server.

7.3.2 Fingerprints

The first step in analyzing whether malware Permission Fingerprints are seen as outliers is to obtain permission fingerprints for all apps in the database. After which, we aggregated the fingerprints by classification of capabilities. The result, Figure 7.5, provides a novel look at malware, Its capabilities, and it's permissions relative to the Android market. The differences in Permission Fingerprints, and therefore capabilities of Malware vs the GPStore total become obvious right away. Malware is far more likely to request access to SMS, particularly reading and writing the database, as well as system operations like disabling the lock screen and reading system logs.

However, further insight is given when looking at the fingerprints of malware with a specific capability. *Spyware* apps demonstrate this quite clearly: They request access to location, permission to keep the device on at all times, record audio and send SMS messages. These capabilities are what would be expected out of spyware. Likewise, *Premium SMS* apps requested access to Send SMS messages quite often[2]. We also observe that *Botnet* malware occasionally accesses system features like disabling the lock screen and reading system logs.

These observations are highly valuable, but things break down when we look at the *Info* category. This permission fingerprint, aside from *Read SMS* and *Write SMS*, does not appear significantly different than the GPStore total, especially when considering the Market Category fluctuations observed earlier in Section 7.2.1.

7.4 By Install Count

As we have shown, all categories of malware but Info Theft Malware show profound differences in permission fingerprints: they are far more likely to access unusual permissions than normal Android apps. However, Info Theft apps did not show such anomalies, making them far easier to hide inside of benign apps - a technique malware writers already use[59]. Thus, if more popular apps have more access to PII, this vector poses a serious threat.

For starters, we note that the metadata in Android Census doesn't give precise download measurements, but rather broad ranges, as seen in Figure 2.3. We plot these in Table 7.5, noting the number of apps in each download range, and Its total percent of all downloads in the market. The top percentile, only 9 apps, accounts for an estimated[3] 11% of all downloads, and apps over 1 million downloads - only 2250 apps in total - account for an estimated 70% of all downloads in the entire Google Play Store.

[2]The reason this number is not 100% is because not all variants of a family of malware known to posses a *Premium SMS* exploit may have that capability

[3]The formula for estimated total downloads was taking the average of the download range, and multiplying it by the number of apps

Download Range	Number of Apps	Estimated Total Downloads	Percent of All Downloads	Aggregate Percentage
100M-500M	9	2.7B	11.78%	11.78%
50M-100M	12	900.0M	3.93%	15.71%
10M-50M	165	4.95B	21.6%	37.31%
5M-10M	279	2.09B	9.13%	46.44%
1M-5M	1785	5.36B	23.36%	69.8%
500K-1M	2010	1.51B	6.58%	76.38%
100K-500K	10483	3.14B	13.72%	90.1%
50K-100K	9363	702.23M	3.06%	93.16%
10K-50K	37925	1.14B	4.96%	98.13%
5K-10K	24872	186.54M	0.81%	98.94%
1K-5K	66587	199.76M	0.87%	99.81%
500-1K	28676	21.51M	0.09%	99.91%
100-500	59616	17.88M	0.08%	99.98%
50-100	23394	1.75M	0.01%	99.99%
10-50	53343	1.6M	0.01%	100.0%
5-10	13720	96.04K	0.0%	100.0%
1-5	30541	91.62K	0.0%	100.0%
0-0	7694	0.0	0.0%	100.0%

Table 7.5: App download statistics from Android Census

Figure 7.6 shows the permission fingerprints of the top download categories on the Google Play Store. The apps in the lower ranges, from 10K-50K, show a fingerprint very similar to the overall plot. However, as the install counts rise, several very key permissions rise as well. *Get Running Apps*, *Camera*, *Record Audio*, *Read Contacts*, and *Wake Lock* all spike in popularity, the more installs the app has. These permissions are especially sensitive PII related permissions. Towards the very top, even more permissions become used, *Read and Write SMS* and *Read Browser History*, which are also extremely common malware permissions. Overall, aside from *Read and Write SMS*, the *Info Theft* malware described above has a surprisingly similar permission fingerprint.

7.5 Conclusion

We have demonstrated several key points in this analysis. First, we demonstrated that Permission Fingerprints often correlate with their expected behavior, but found key instances where these did not, especially with respect to PII. We then demonstrated that different classes of malware have permission fingerprints that correlate strongly with their expected behavior, but vary heavily from the rest of the GPStore. However, we noted that *SMS* malware only needed one key permission to operate - one that was ultimately not terribly uncommon in many categories, and Info Theft apps had a permission fingerprint that does not ultimately stick out in the GPStore as a whole. When analyzing the top apps, we found many opportunities for Info Theft malware to imitate their permission fingerprints.

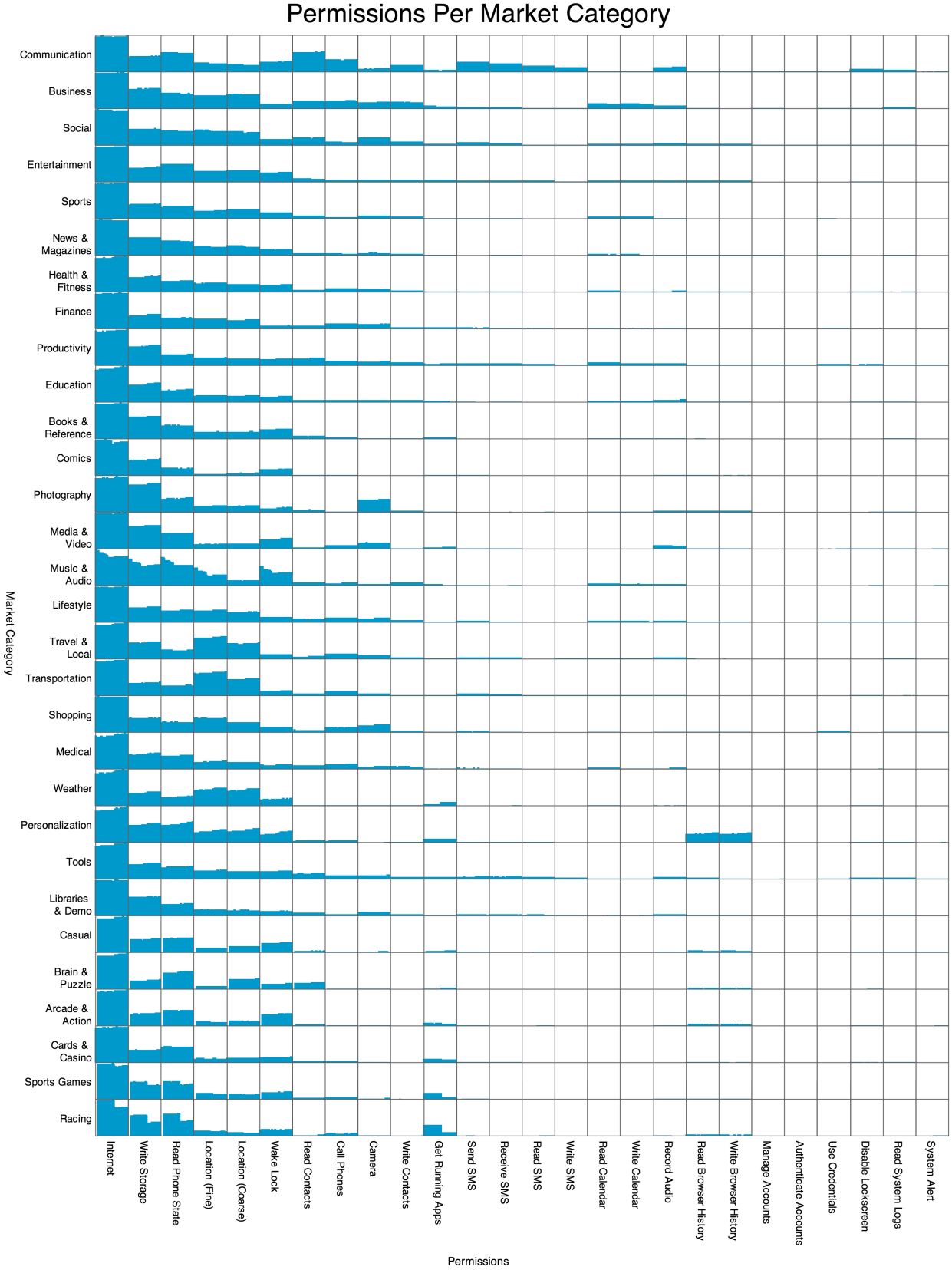

Figure 7.3: Permissions used, as a fraction of total in that category

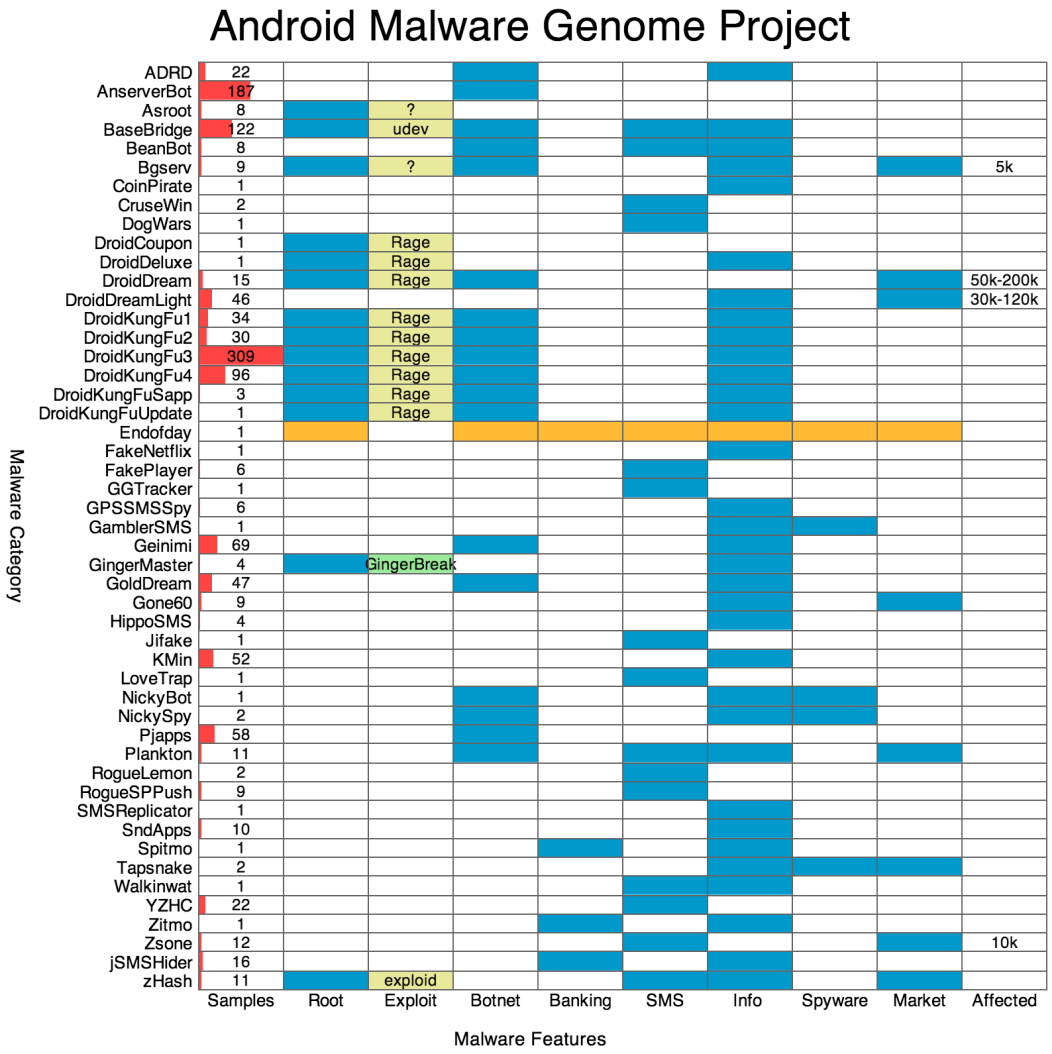

Figure 7.4: A summary of the malware families found in the Android Malware Genome Project, see Figure 7.4 for explanations of the classes

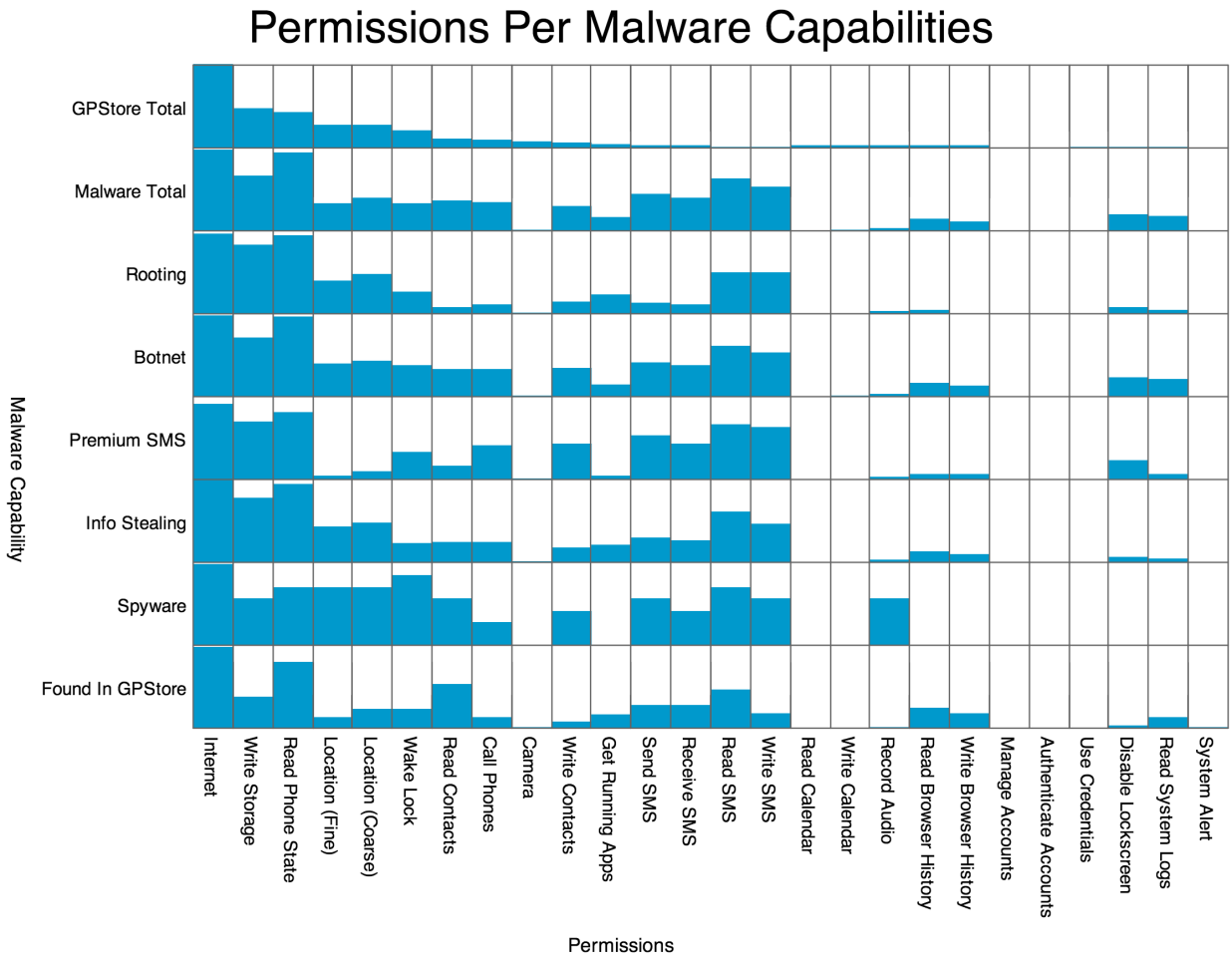

Figure 7.5: Permission Fingerprints of malware with different capabilities, compared to the GPStore total

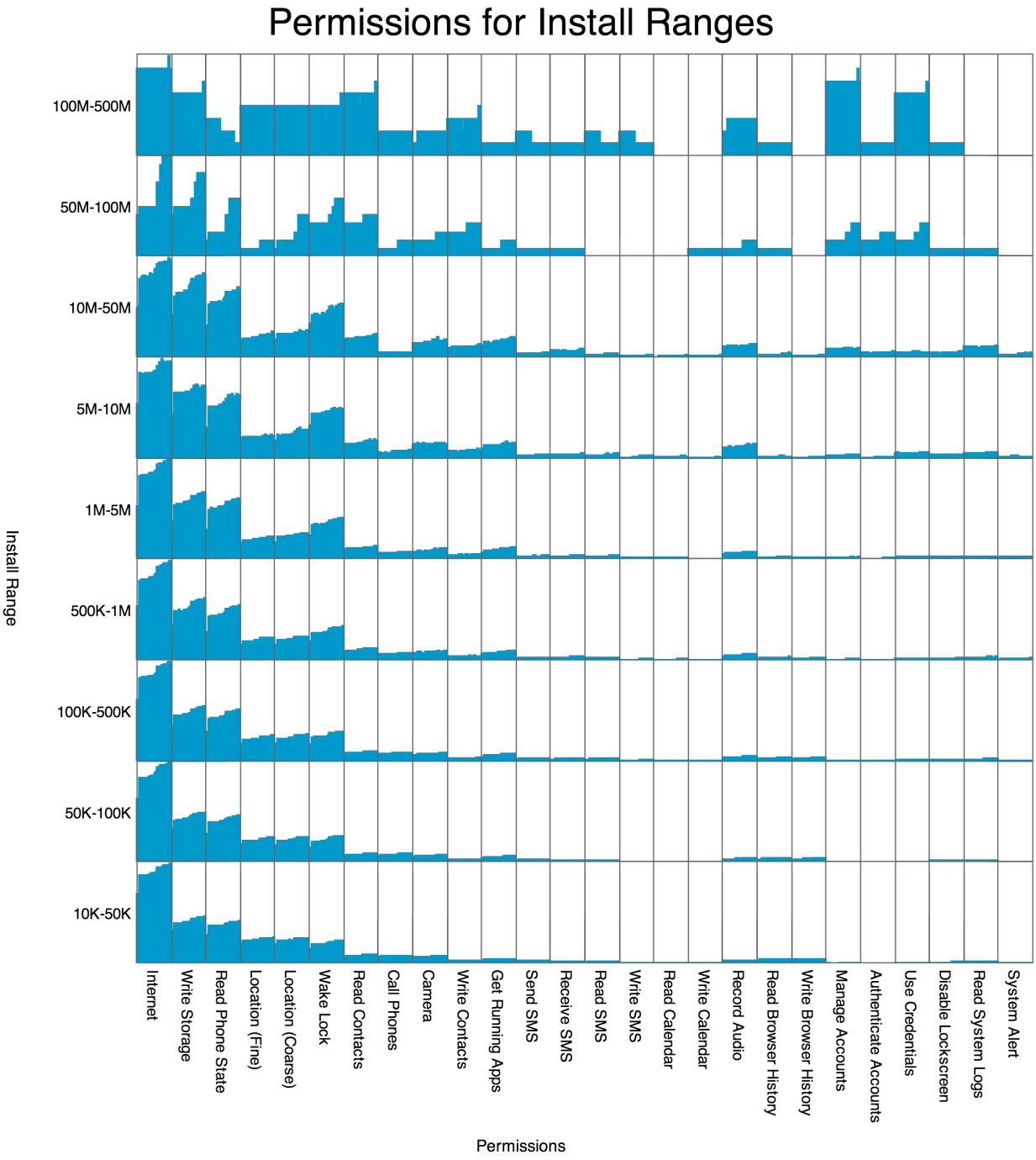

Figure 7.6: Permission Fingerprints of apps with different ranges of install counts

Chapter 8

Evaluation

With AndroMEDA, we attempt to build on top of the Android Permission system, to do a better job at enforcing the User-App Agreement. The main reasons why the Permission System failed to differentiate between malware and normal software can be seen as a lack of context and understanding of use. When an app requests a permission, it is granted to the app regardless of any context - at any time, and regardless of user consent. The data, after being requested, ultimately can be manipulated and transmitted to any party without user consent.

Projects like TaintDroid[21] have begun to address the flow of personal data, which aids in the user understanding the use of data. However, much more can be done to address both context and use, which AndroMEDA addresses. By instrumenting API calls, AndroMEDA can both inspect context use of personal data as well as other important system actions. By presenting this normally hidden information to the user, AndroMEDA provides them with a feedback loop to evaluate whether the User-App Agreement has been broken.

8.1 Existing Malware Datasets

To test the effectiveness of AndroMEDA at detecting malicious behavior, we begin with testing apps from the Android Malware Genome Project[67], the most comprehensive academic malware dataset. Unfortunately, as show in in Figure 8.1, we found only 31 (2.4%) of the nearly 1300 samples were designed for Android 2.3 and above, when Android fixed many rootkits, and not a single one targeted Android 4.0 or above, when Android took steps to fix SMS related malware. When looking for malware that runs on Android 2.3 and above that do not use a root exploit, we find only 4 samples - 0.3% - fit.

Of these 4 examples, one, *RogueSPPush*, sends premium SMS messages, but does not request *Send SMS*, rendering it ineffective. The 3 remaining samples, *DogWars*, *GoldDream* and *DroidDreamLight* were Info Theft malware. We found *DroidDreamLight* to be less than worthwhile for testing, due to its rather benign nature: the worst action it performs is uploading the device's IMEI to a remote server. Finally, *GoldDream* has botnet capabilities, making it difficult to test, due to the lack of an existing botnet.

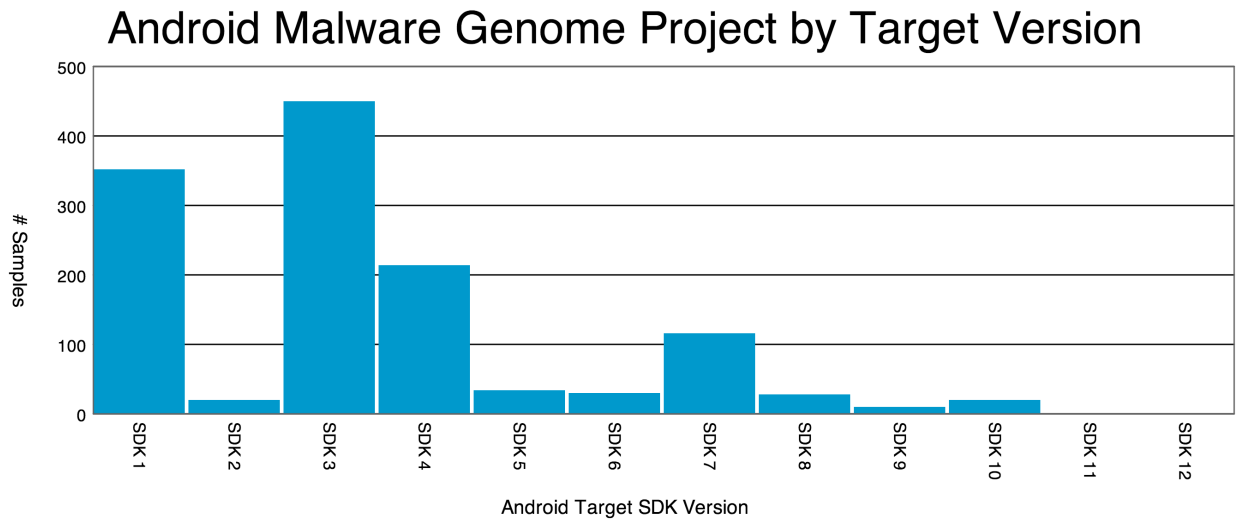

Figure 8.1: Malware from the Android Malware Genome Project by Android Version

The remaining app, *DogWars*, makes a good testing app. Made as a "hacktivist" app in protest of an existing app[12], *DogWars* repackages a game with code to send SMS messages to all contacts on the next boot. The screenshots in Figure 8.2 show how AndroMEDA clearly identifies when DogWars accesses contacts and sends messages, and allows the user to identify it as malware.

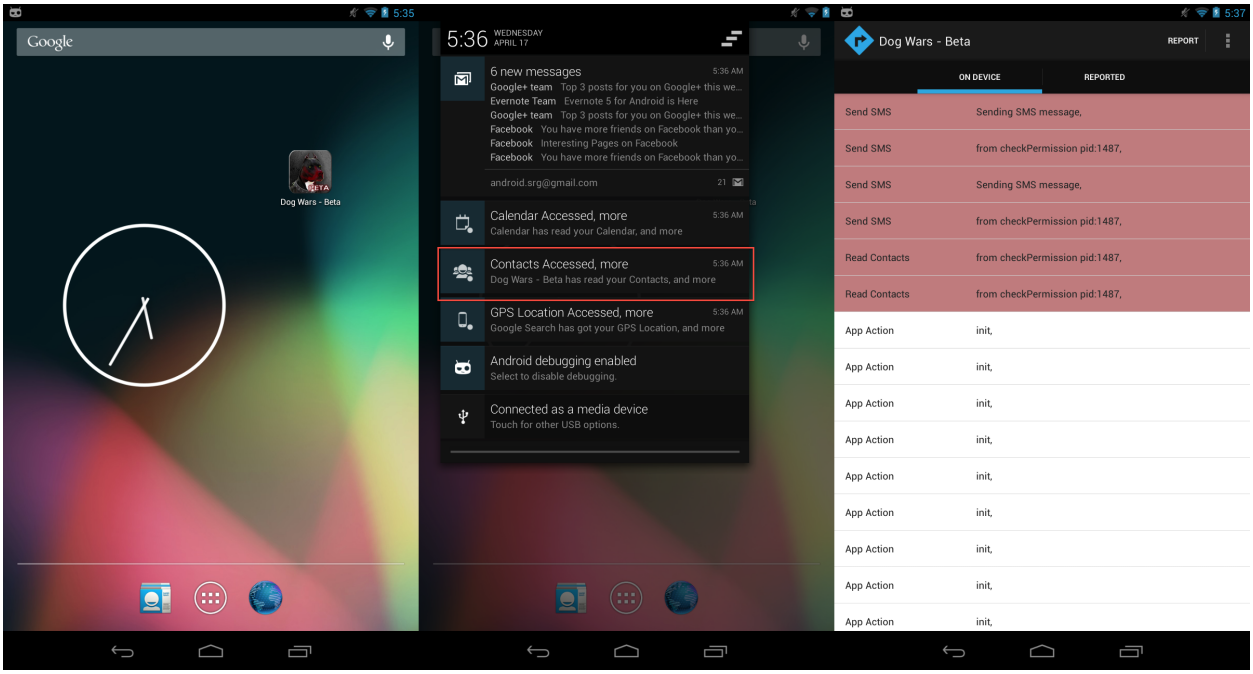

Figure 8.2: AndroMEDA detecting the *DogWars* Malware (annotated in red)

We have demonstrated AndroMEDA's ability to identify malware in existing datasets, but in the process we highlighted the shortcomings of the current Android malware datasets. To test a future-malware oriented framework, it would be ideal to have more sophisticated malware. Rootkits and Premium SMS malware have been addressed by recent versions of Android, so we focus on what we believe to be the future of malware on Android: Sophisticated Info Theft Apps. The concept of hiding Info Theft Malware inside of benign apps, designed to silently steal PII and perform other unwanted operations, we have dubbed IncognitoWare. Since our analysis of Permissions in the GPStore has shown that popular apps tend to request a large amount of PII-related permissions, we can fit them all within the existing Permission Fingerprint.

8.2 IncognitoWare Dataset

IncognitoWare has recently became one of the most popular forms of mobile malware[42]. FakeInst, discussed in Chapter 4.5 was a repackaged version of Instagram[32] that sent premium SMS messages on start. While admittedly basic, more complex versions, like FakeAngry[65] have been found, imitating popular games, while in the background stealing PII, installing a rootkit and joining a botnet.

We introduce a novel set of research IncognitoWare, as a representative sample of current and future mobile malware techniques. Our first example sends as much PII as it can find to a remote server, the second silently monitors the phone in the background. We chose not to include a Premium SMS Malware sample, despite its popularity[42], due to it being addressed in the latest version of Android. By analyzing our framework with this dataset, we hope to demonstrate the effectiveness of addressing the UAA as a main route to detecting malware.

Creating IncognitoWare is straightforward. First, an exploit is designed, and coded as its own app. Second, the *apktool*[10] utility decompiles any APK file into a set of resources and *smali* files - decompiled Dalivk bytecode. From there, the *apktool* utility is used again on the exploit, and the *smali* code trees are merged. The exploit entry points are then placed inside of the host app's code, and *apktool* rebuilds the project back into an APK file. This APK is unsigned, and requires the malware writer to resign it. This mismatched signature makes the APK unsuitable for uploading to the Google Play Store, but by changing the Android package to something slightly different, it's suitable for deployment in the Google Play Store, or 3rd party markets.

8.3 Info Theft IncognitoWare

Our first example of next-generation IncognitoWare is simple: embed PII stealing code into any app, but only execute it after the user has logged in. This act of logging in is sufficient to bypass automated monitoring tools like Google Bouncer. Only executing the code after the user has performed an action also creates a plausible scenario where the

situation might have been intended. Ultimately, however, since nothing is presented to the user, such an action is a clear violation of the UAA, and is seen as malicious. Furthermore, we stay within the permission fingerprint of the original software, meaning not only is it invisible when installing, but blocking the permission outright, or feeding it fuzzed data, would ultimately result in legitimate actions being interfered with. This simple example is powerful enough to steal nearly all highly valuable PII from a device, but inconspicuous enough to be undetected, and legitimate enough to be unblocked.

AndroMEDA, however, can easily detect this example, seen in Figure 8.3. After the user logs in, AndroMEDA immediately alerts the user that the PII has been read. The user then reviews Figure 8.5 and decides if its actions break the UAA, and if the app should be trusted with the actions. By comparison, the logs of the untainted version of the app can be seen in Figure 8.4. It's worth noting that the logs in Figure 8.4 and 8.5 were not generated by the current companion app, but offline, with additional context added - describing what the user was doing at the time of specific actions, and that these visualizations are a focus of future work.

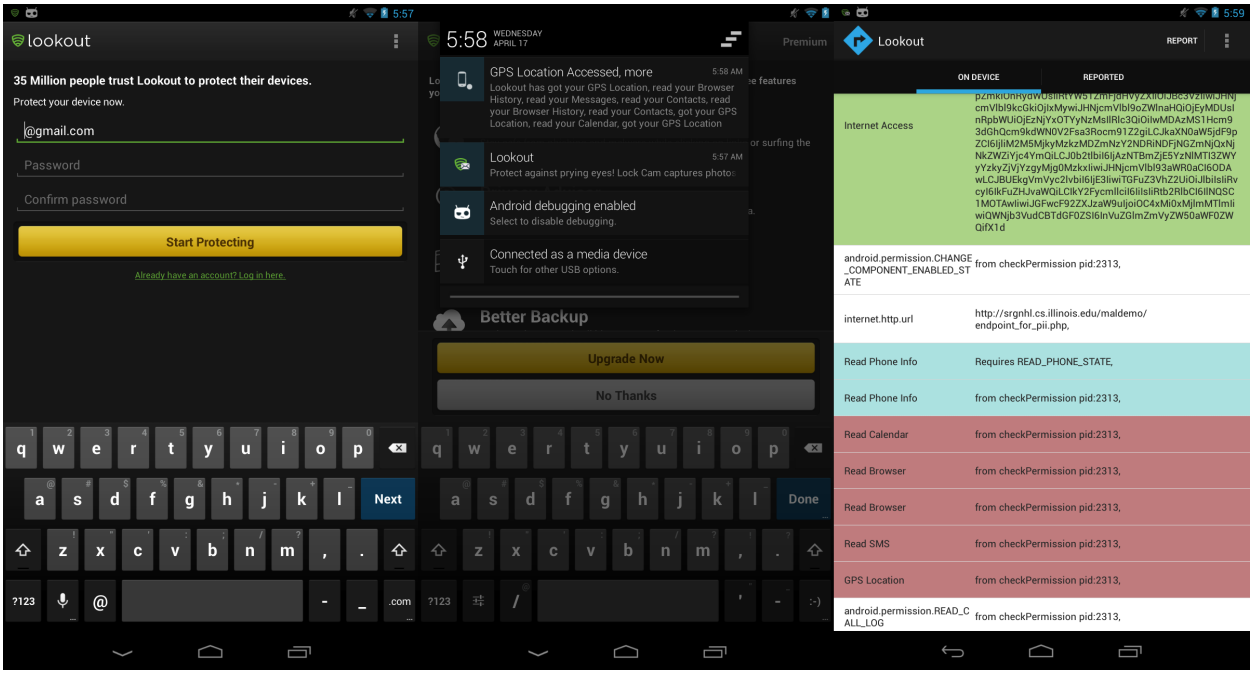

Figure 8.3: AndroMEDA detecting the Info Theft IncognitoWare embedded within a security app

8.4 Spyware IncognitoWare

Our next example of sophisticated IncognitoWare is designed to continuously spy on the user, while staying within the capabilities of a trusted app. We first find an app that has all the requirements for spyware: access to location,

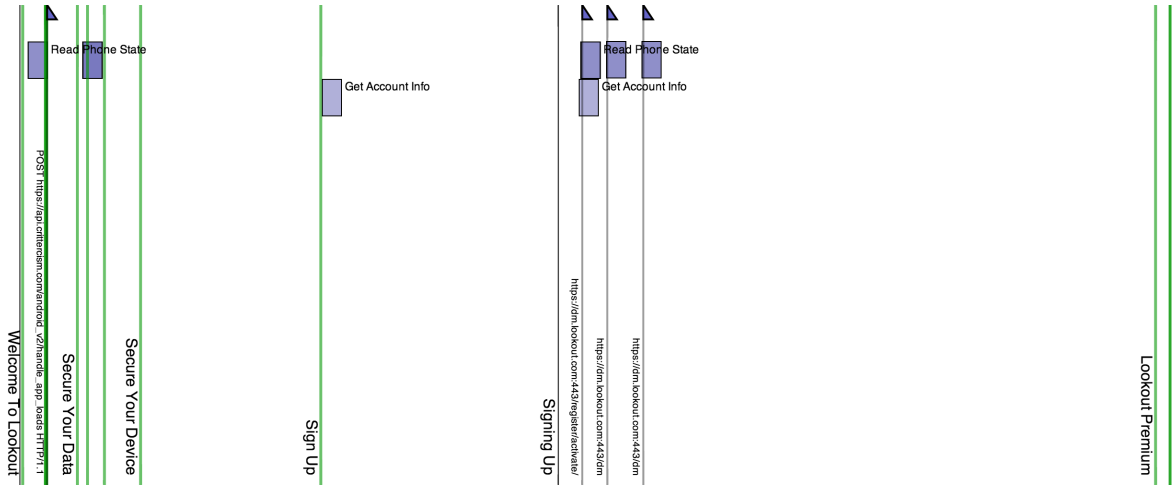

Figure 8.4: AndroMEDA logs of the normal version of the security app

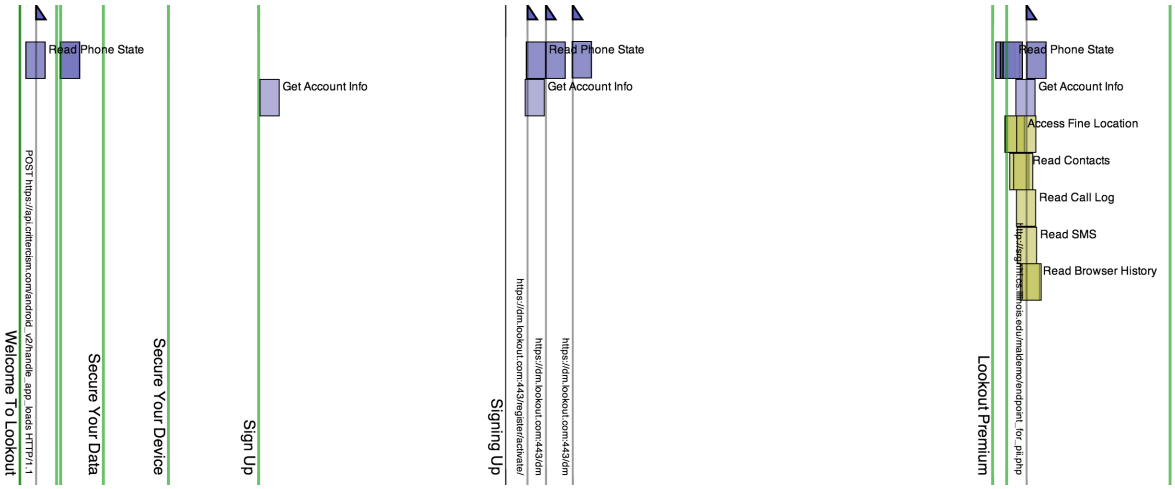

Figure 8.5: AndroMEDA logs of Info Theft IncognitoWare embedded within a security app

internet, wake lock, and starting at boot. We then introduce our malware, which is triggered an arbitrary amount of minutes after the first startup - 5 minutes is long enough to evade Google Bouncer[36]. After that, the device wakes up every 5 minutes, gathers a location fix, and sends that information to a remote location. As before, many examples exist of apps that use similar behavior for non malicious purposes, making policies to guard against it difficult.

Once again, AndroMEDA can alert the user of this kind of behavior easily. Figure 8.6 shows the user installing the app, starting it up, going to the home screen, and eventually noticing the suspicious behavior when the phone is idling. The user then inspects the logs for this app, Figure 8.7 (with context added), finds this to be a continuous occurrence, and decides that the app is malware. Compared with Figure 8.9, the app is clearly accessing location in a suspicious pattern, and the usage pattern is the key indicator. However, when compared to Figure 8.8, the main indicator is the

address the information is going towards - an address not associated with the main app. The normal logs in Figure 8.8 may break UAA for some users - the app logs the user's location in the background and sends it to a remote server. Overall, this highlights the ability of AndroMEDA to capture both the context and use of permissions - the patterns of access provide context, and the network locations provide use.

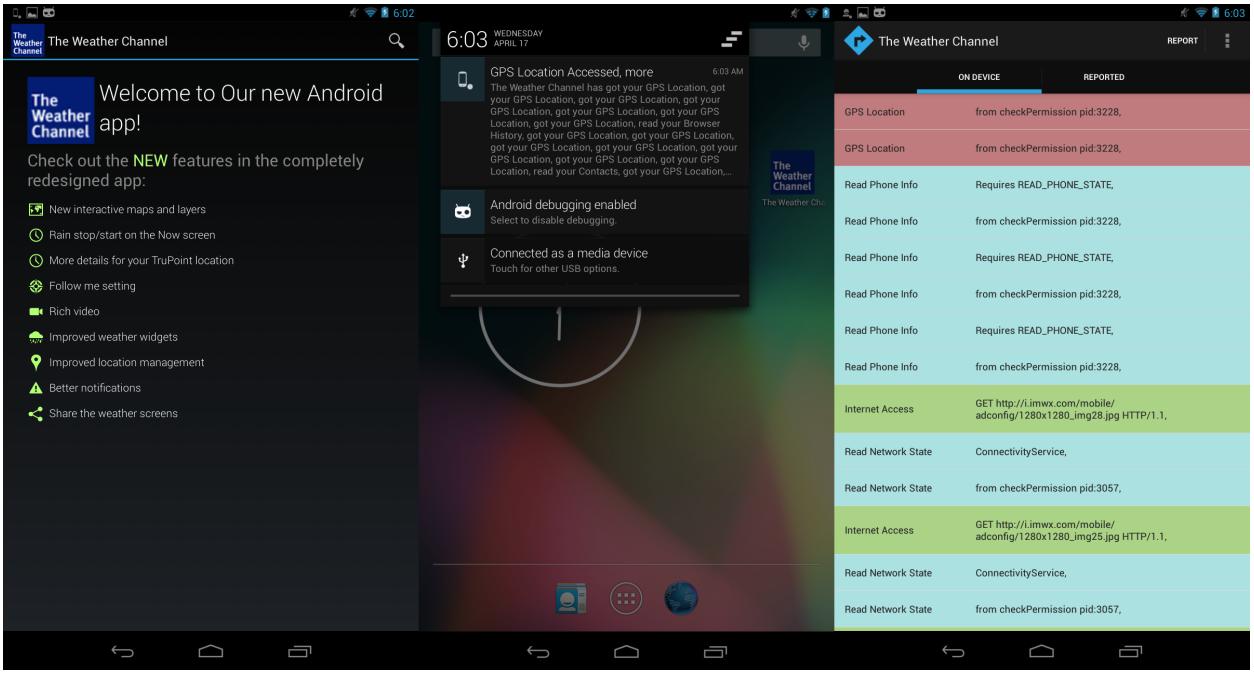

Figure 8.6: AndroMEDA detecting the Spyware IncognitoWare embedded within a weather app

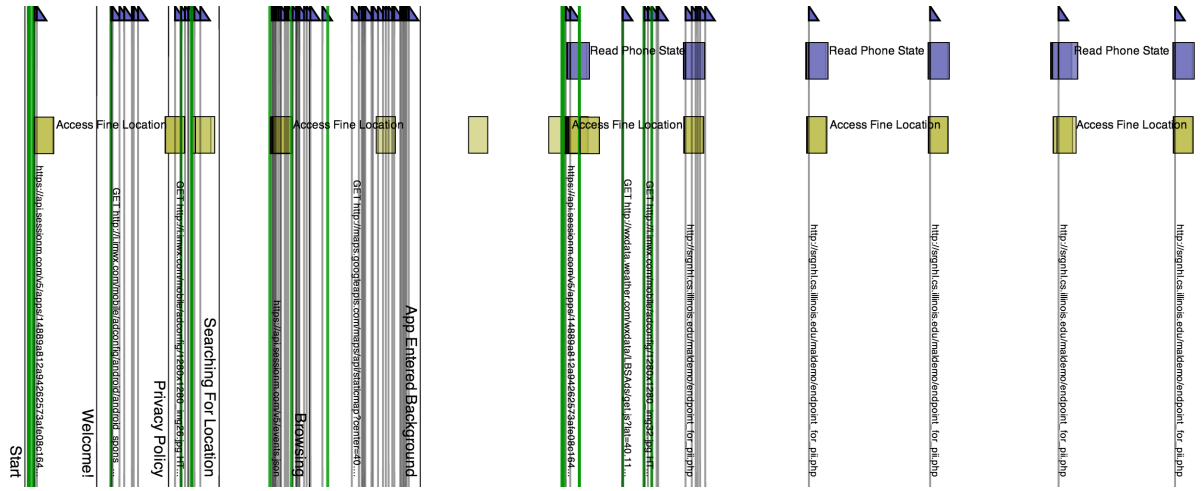

Figure 8.7: AndroMEDA logs of Spyware IncognitoWare embedded within a weather app

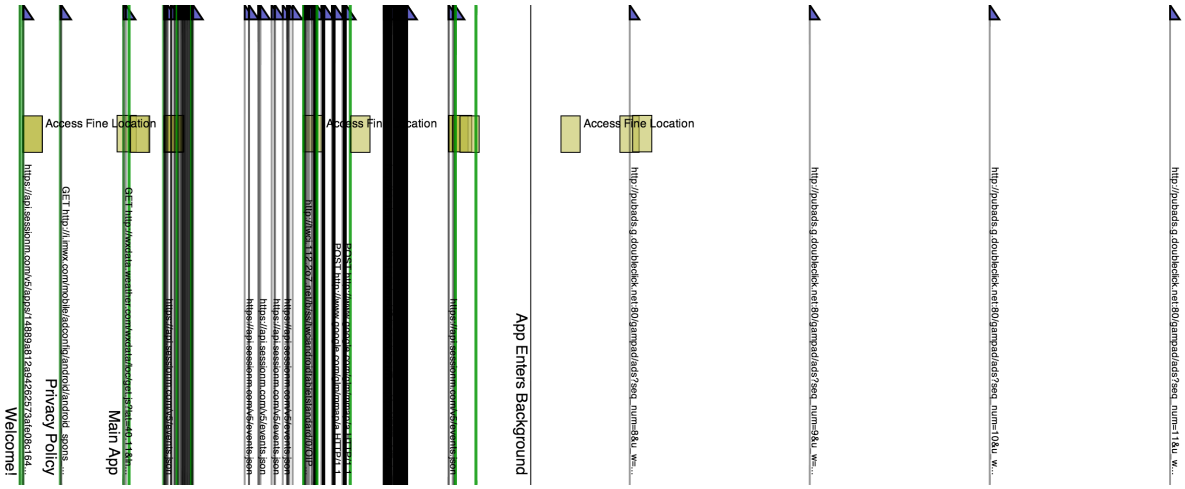

Figure 8.8: AndroMEDA logs of a normal version of a weather app, when the user has consented to location gathering

Figure 8.9: AndroMEDA logs of a normal version of a weather app, when the user has not consented to location gathering

8.5 Companion App

When the user reviews the logs, they have the option to report suspicious behavior, giving a description of what the user was doing, and what the suspicious behavior was. These reports are collected in a centralized database. The companion app also listens for when the user installs new applications, and checks the same database to see if there are any existing reports. If the number of reports passes a threshold, the companion app will notify the user, giving them the ability to review what other people have reported about the apps. These features - being notified of suspicious apps and reviewing them - does not require the AndroMEDA extensions, and can be installed on any Android device. This enables a small set of users to alert a much larger population.

8.6 Conclusion

Overall, we have demonstrated that the ability to log and visualize app behavior can lead to an increased ability to detect malware. Our logs and visualizations show clear differences between actions that fall within the User-App Agreement, and actions that do not, even being able to visualize actions that may break some user's UAA but not others. Being able to report when an app breaks the UAA, and having that report spread to other users, enables AndroMEDA to become a community effort, further increasing its effectiveness.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

AndroMEDA helps users understand the context in which their Personally Identifiable Information is used, which allows them to make more informed decisions on whether an app is action maliciously or not. In this paper, we introduced 4 key items:

9.1.1 User-App Agreement

We analyzed the current Android security framework: The Permission System, and found its main flaws were its lack of addressing context and use, which we generalize into the User-App Agreement - a framework for consenting and trusting specific actions an app may take. Whereas Android Permissions exceeded at defining general capabilities of an app, and these capabilities go a long way in shaping the User-App Agreement, they fail at addressing the context in which the permissions are used, and what they are used for.

9.1.2 Android Census

To perform a full analysis of the current state of Android Permissions, we use a novel dataset, Android Census. By analyzing more than 80% of apps in the Google Play Store, we are able to better understand the interrelationship of permissions and expected behavior. We produce key insights as to the popularity of apps vs their PII permissions, and when apps deviate from their expected behavior - potentially violating the User-App Agreement. We then analyze a comprehensive malware dataset using the same techniques, and find that many types of malware can be identified purely by its permission fingerprint. We conclusively show the connection between Permissions and Expected Behavior are present, but not strong enough to differentiate between Info Theft Malware and many popular apps.

9.1.3 AndroMEDA

Building off the concept of the User-App Agreement, we introduce AndroMEDA. Key parts of the User-App Agreement were previously unnoticeable to the user until AndroMEDA. By giving the user more information on the context and use of permissions, they can evaluate whether they trust those actions, and ultimately whether the app is acting maliciously or not. After untrusted behavior is spotted, the actions can be reported, and knowledge can be spread to all users. All of this makes users more aware of app behavior, and helps mitigate Info Theft Malware on Android.

9.1.4 IncognitoWare

We highlighted the need for more modern Android malware datasets. To address this, we introduced a dataset of IncognitoWare - repackaged trusted apps with additional malicious behavior embedded. We believe this is a more comprehensive look at Android security because it highlights the need to understand context in order to identify malicious behavior.

9.2 Future Work

AndroMEDA is, ultimately, not a silver bullet at detecting all Android malware. Projects like TaintDroid and TISSA provide functionality that would greatly enhance the data gathering abilities and response options of AndroMEDA. Integrating TISSA would allow users or AndroMEDA to temporally block access to sensitive data, while not blocking access to the same data at a later time, when the user trusts the action. Overall, adding more API instrumenting would prove useful, as projects like TapLogger provide future challenges.

A main focus of AndroMEDA is providing a feedback loop to the user, and to that end, visualization is an important area of future work. Visualizing the wealth of information in a concise way that avoids user fatigue is a main challenge, and subject to future study.

AndroMEDA could also benefit greatly from the probabilistic modeling of pBMDs and Crowdroid, in correlating user action with permission behavior. These would not replace the need to alert the user, but rather be able to better dictate when to put different classes of alerts to the user, as ultimately the decision of what is malware is up to them.

The wealth of data in Android Census was also not fully explored. We are currently interested in seeing if specific keywords in user reviews correlate with malicious software, or other problematic apps. Many more areas of metadata, like the description, developer, etc, can be further explored, to see if it gives additional insight into the nature of malware on Android.

Finally, the concept of the User-App Agreement introduced in this paper can be expanded upon greatly. User Studies of trust in specific actions can greatly increase our understanding of what actions users tend to trust, and when

they are untrustworthy. These same user studies could also help show the effectiveness of AndroMEDA in spotting Info Theft Malware.

Appendix A

A.1 Android Permissions

Name	ID	Severity	3rd Party Usable
Bind Wallpaper	*android.permission.BIND_WALLPAPER*	signatureOrSystem	no
View Google Services	*com.google.android.providers.gsf.permission.-READ_GSERVICES*	none	yes
Force Back	*android.permission.FORCE_BACK*	signatureOrSystem	no
Read Calendar	*android.permission.READ_CALENDAR*	dangerous	yes
Read Frame Buffer	*android.permission.READ_FRAME_BUFFER*	signature	no
NFC	*android.permission.NFC*	dangerous	yes
Read Sync Stats	*android.permission.READ_SYNC_STATS*	none	yes
Battery Stats	*?android.permission.BATTERY_STATS*	none	yes
Internet	*android.permission.INTERNET*	dangerous	yes
Change Config	*android.permission.CHANGE_CONFIGURATION*	dangerous	yes
Google Auth - Docs	*com.google.android.googleapps.permission.-GOOGLE_AUTH.writely*	dangerous	yes
Test	*android.permission.HARDWARE_TEST*	signature	no
Read Google Contacts	*com.google.android.googleapps.permission.-GOOGLE_AUTH.cp*	dangerous	yes
Read GTalk	*com.google.android.providers.talk.permission.READ_ONLY*	dangerous	yes
Google Auth - Calendar	*com.google.android.googleapps.permission.-GOOGLE_AUTH.cl*	dangerous	yes
Bind Input	*android.permission.BIND_INPUT_METHOD*	signature	no
Set Time Zone	*android.permission.SET_TIME_ZONE*	dangerous	yes
Access Cache	*android.permission.ACCESS_CACHE_FILESYSTEM*	signatureOrSystem	no
Write Sync Settings	*android.permission.WRITE_SYNC_SETTINGS*	dangerous	yes

Change Data Settings	*android.permission.CHANGE_BACKGROUND_DATA_SETTING*	signature	no
Write Google Services	*android.permission.WRITE_GSERVICES*	signatureOrSystem	no
Inject Events	*android.permission.INJECT_EVENTS*	signature	no
Bind Device Admin	*android.permission.BIND_DEVICE_ADMIN*	signature	no
Force Stop	*android.permission.FORCE_STOP_PACKAGES*	signature	no
Write GTalk	*com.google.android.providers.talk.permission.WRITE_ONLY*	dangerous	yes
Write Secure Settings	*android.permission.WRITE_SECURE_SETTINGS*	signatureOrSystem	no
Call All Numbers	*android.permission.CALL_PRIVILEGED*	signatureOrSystem	no
Broadcast App Removed	*android.permission.BROADCAST_PACKAGE_REMOVED*	signatureOrSystem	no
System Alert	*android.permission.SYSTEM_ALERT_WINDOW*	dangerous	yes
Location - Extra	*android.permission.ACCESS_LOCATION_EXTRA_COMMANDS*	none	yes
Brick	*android.permission.BRICK*	signature	no
Dump System	*android.permission.DUMP*	signatureOrSystem	no
Shutdown	*android.permission.SHUTDOWN*	signature	no
Change WiFi	*android.permission.CHANGE_WIFI_STATE*	dangerous	yes
Receive SMS	*android.permission.RECEIVE_SMS*	dangerous	yes
Modify Phone	*android.permission.MODIFY_PHONE_STATE*	signatureOrSystem	no
Read Attachment	*com.google.android.gm.permission.-READ_ATTACHMENT_PREVIEW*	signature	no
Manage Accounts	*android.permission.ACCOUNT_MANAGER*	signature	no
Read GMail	*com.google.android.gm.permission.READ_GMAIL*	signature	no
Animation Scale	*android.permission.SET_ANIMATION_SCALE*	dangerous	yes
Set Process Limit	*android.permission.SET_PROCESS_LIMIT*	dangerous	yes
Move Package	*android.permission.MOVE_PACKAGE*	signatureOrSystem	no
Google Auth - Other	*com.google.android.googleapps.permission.-GOOGLE_AUTH.OTHER_SERVICES*	dangerous	yes
Debug App	*android.permission.SET_DEBUG_APP*	?	no
Install DRM	*android.permission.INSTALL_DRM*	none	yes
Bluetooth	*android.permission.BLUETOOTH*	dangerous	yes
Camera	*android.permission.CAMERA*	dangerous	yes
Set Wallpaper Hints	*android.permission.SET_WALLPAPER_HINTS*	none	yes
Reboot	*android.permission.REBOOT*	signatureOrSystem	no
Broadcast WAP	*android.permission.BROADCAST_WAP_PUSH*	signature	no

Google Auth - Maps	*com.google.android.googleapps.permission.-GOOGLE_AUTH.local*	dangerous	yes
View Network	*android.permission.ACCESS_NETWORK_STATE*	none	yes
Status Bar	*android.permission.STATUS_BAR*	signatureOrSystem	no
Write User Dictionary	*android.permission.WRITE_USER_DICTIONARY*	none	yes
Read Browser History	*com.android.browser.permission.-READ_HISTORY_BOOKMARKS*	dangerous	yes
Access DRM	*android.permission.ACCESS_DRM*	signature	no
Record Audio	*android.permission.RECORD_AUDIO*	dangerous	yes
Write Contacts	*android.permission.WRITE_CONTACTS*	dangerous	yes
Send Gmail	*com.google.android.gm.permission.AUTO_SEND*	signature	no
Control Location	*android.permission.CONTROL_LOCATION_UPDATES*	signatureOrSystem	no
Bind Widgets	*android.permission.BIND_APPWIDGET*	signatureOrSystem	no
Send Linux Signals	*android.permission.SIGNAL_PERSISTENT_PROCESSES*	dangerous	yes
Install Location Provider	*android.permission.INSTALL_LOCATION_PROVIDER*	signatureOrSystem	no
Google Auth - Drive	*com.google.android.googleapps.permission.-GOOGLE_AUTH.wise*	dangerous	yes
Start On Boot	*android.permission. RECEIVE_BOOT_COMPLETED*	none	yes
Clear Phone	*android.permission.MASTER_CLEAR*	signatureOrSystem	no
Read Input State	*android.permission.READ_INPUT_STATE*	signature	no
Internal System UI	*android.permission.INTERNAL_SYSTEM_WINDOW*	signature	no
Manage App Token	*android.permission.MANAGE_APP_TOKENS*	signature	no
Access Email	*com.android.email.permission.ACCESS_PROVIDER*	?	no
Subscribe Feeds	*android.permission.WRITE_SETTINGS*	dangerous	yes
SIP	*android.permission.USE_SIP*	dangerous	yes
Google Auth - AppEngine	*com.google.android.googleapps.permission.-GOOGLE_AUTH.ah*	dangerous	yes
Write APM	*android.permission.WRITE_APN_SETTINGS*	dangerous	yes
Access Surface	*android.permission.ACCESS_SURFACE_FLINGER*	signature	no
Factory Test	*android.permission.FACTORY_TEST*	signature	no
Google Auth - Gmail	*com.google.android.googleapps.permission.-GOOGLE_AUTH.mail*	dangerous	yes
Read System Logs	*android.permission.READ_LOGS*	dangerous	yes
Process Outgoing Calls	*android.permission.PROCESS_OUTGOING_CALLS*	dangerous	yes

Update Device Stats	*android.permission.UPDATE_DEVICE_STATS*	signature	no
Write Calendar	*android.permission.WRITE_CALENDAR*	dangerous	yes
Google Auth - Youtube	*com.google.android.googleapps.permission.-GOOGLE_AUTH.youtube*	dangerous	yes
Read Feeds	*android.permission.SUBSCRIBED_FEEDS_READ*	none	yes
Manage Accounts	*android.permission.MANAGE_ACCOUNTS*	dangerous	yes
Send SMS	*android.permission.SEND_SMS*	dangerous	yes
Google Auth - Blogger	*com.google.android.googleapps.permission.-GOOGLE_AUTH.blogger*	dangerous	yes
Mock Location	*android.permission.ACCESS_MOCK_LOCATION*	dangerous	yes
Change WiFi Multicast	*android.permission.CHANGE_WIFI_MULTICAST_STATE*	dangerous	yes
Access Passwords	*com.google.android.googleapps.permission.-ACCESS_GOOGLE_PASSWORD*	signatureOrSystem	no
Google Auth - All	*com.google.android.googleapps.permission.-GOOGLE_AUTH.ALL_SERVICES*	dangerous	yes
Write SMS	*android.permission.WRITE_SMS*	dangerous	yes
Get Running Apps	*android.permission.GET_TASKS*	dangerous	yes
Delete Packages	*android.permission.DELETE_PACKAGES*	signatureOrSystem	no
Access Checkins	*android.permission.ACCESS_CHECKIN_PROPERTIES*	signatureOrSystem	no
Set Preferred Apps	*android.permission.SET_PREFERRED_APPLICATIONS*	signature	no
Set Time	*android.permission.SET_TIME*	signatureOrSystem	no
Vibrate	*android.permission.VIBRATE*	none	yes
Diagnostic	*android.permission.DIAGNOSTIC*	signature	no
Call Phones	*android.permission.CALL_PHONE*	dangerous	yes
Flashlight	*android.permission.FLASHLIGHT*	none	yes
Read Phone State	*android.permission.READ_PHONE_STATE*	dangerous	yes
Location (Coarse)	*android.permission.ACCESS_COARSE_LOCATION*	dangerous	yes
Clear App Data	*android.permission.CLEAR_APP_USER_DATA*	signature	no
Broadcast SMS	*android.permission.BROADCAST_SMS*	signatureOrSystem	no
Kill Background	*android.permission.KILL_BACKGROUND_PROCESSES*	none	yes
Stop App Switching	*android.permission.STOP_APP_SWITCHES*	signature	no
Access WiFi	*android.permission.ACCESS_WIFI_STATE*	none	yes
Receive SMS	*android.permission.RECEIVE_MMS*	dangerous	yes
Wake Lock	*android.permission.WAKE_LOCK*	dangerous	yes

Write Browser History	*com.android.browser.permission.-WRITE_HISTORY_BOOKMARKS*	dangerous	yes
Delete Cache Files	*android.permission.DELETE_CACHE_FILES*	signatureOrSystem	no
View Google Auth	*com.google.android.googleapps.permission.GOOGLE_AUTH*	none	yes
Read Accounts	*android.permission.GET_ACCOUNTS*	none	yes
Change Network	*android.permission.CHANGE_NETWORK_STATE*	dangerous	yes
Read Sync Settings	*android.permission.READ_SYNC_SETTINGS*	none	yes
Disable Lockscreen	*android.permission.DISABLE_KEYGUARD*	dangerous	yes
Write Gmail	*com.google.android.gm.permission.WRITE_GMAIL*	signature	no
Use Credentials	*android.permission.USE_CREDENTIALS*	dangerous	yes
Write Feeds	*android.permission.SUBSCRIBED_FEEDS_WRITE*	dangerous	yes
Kill Background Apps	*? android.permission.KILL_BACKGROUND_PROCESSES*	none	yes
Change Components	*android.permission.CHANGE_COMPONENT_ENABLED_STATE*	signature	no
Backup	*android.permission.BACKUP*	signatureOrSystem	no
Google Auth - Finance	*com.google.android.googleapps.permission.-GOOGLE_AUTH.finance*	dangerous	yes
Expand Status Bar	*android.permission.EXPAND_STATUS_BAR*	none	yes
Bluetooth Admin	*android.permission.BLUETOOTH_ADMIN*	dangerous	yes
Location (Fine)	*android.permission.ACCESS_FINE_LOCATION*	dangerous	yes
Youtube Username	*com.google.android.googleapps.permission.-GOOGLE_AUTH.YouTubeUser*	dangerous	yes
Persist Activity (dep.)	*android.permission.PERSISTENT_ACTIVITY*	DEPRECATED	no
Reorder Tasks	*android.permission.REORDER_TASKS*	?	no
Receive WAP Push	*android.permission.RECEIVE_WAP_PUSH*	dangerous	yes
Receive C2DM	*com.google.android.c2dm.permission.RECEIVE*	none	yes
Set Wallpaper	*android.permission.SET_WALLPAPER*	none	yes
Google Auth - Picasa	*com.google.android.googleapps.permission.-GOOGLE_AUTH.lh2*	dangerous	yes
Read User Dictionary	*android.permission.READ_USER_DICTIONARY*	dangerous	yes
Write Storage	*android.permission.WRITE_EXTERNAL_STORAGE*	dangerous	yes
Get Package Size	*android.permission.GET_PACKAGE_SIZE*	none	yes
Install Packages	*android.permission.INSTALL_PACKAGES*	signatureOrSystem	no
Authenticate Accounts	*android.permission.AUTHENTICATE_ACCOUNTS*	dangerous	yes
Set Alarm	*com.android.alarm.permission.SET_ALARM*	none	yes

Google Auth	*com.google.android.googleapps.permission.-GOOGLE_AUTH.grandcentral*	dangerous	yes
Read Contacts	*android.permission.READ_CONTACTS*	dangerous	yes
CDMA Provisioning	*android.permission.PERFORM_CDMA_PROVISIONING*	signatureOrSystem	no
Modify Audio	*android.permission.MODIFY_AUDIO_SETTINGS*	dangerous	yes
Set Orientation	*android.permission.SET_ORIENTATION*	signature	no
Set Activity Watcher	*android.permission.SET_ACTIVITY_WATCHER*	signature	no
Read SMS	*android.permission.READ_SMS*	dangerous	yes
Broadcast Sticky	*android.permission.BROADCAST_STICKY*	none	yes
Mount Filesystems	*android.permission.MOUNT_FORMAT_FILESYSTEMS*	dangerous	yes
Clear App Cache	*android.permission.CLEAR_APP_CACHE*	dangerous	yes
Mount Filesystems	*android.permission.MOUNT_UNMOUNT_FILESYSTEMS*	dangerous	yes

Table A.1: All Android Permissions in the Google Play Store

References

[1] Address Book Programming Guide for iOS - User Interaction: Prompting for and Displaying Data. http://developer.apple.com/library/ios/#documentation/ContactData/Conceptual/AddressBookProgrammingGuideforiPhone/Chapters/UI_Controllers.html#//apple_ref/doc/uid/TP40007744-CH5-SW1.

[2] Android Architecture Overview. http://developer.android.com/images/system-architecture.jpg.

[3] Android Census. http://mosyg2.cs.uiuc.edu/androidcensus/.

[4] Android Dashboards. http://developer.android.com/about/dashboards/index.html.

[5] Android Developers - Content Providers. http://developer.android.com/guide/topics/providers/content-providers.html.

[6] Android Developers - Intents. http://developer.android.com/reference/android/content/Intent.html.

[7] Android Jellybean 4.2 Security. http://developer.android.com/about/versions/jelly-bean.html#42-security.

[8] Android Open Source Project. http://source.android.com/.

[9] Android Pit. http://www.androidpit.com/.

[10] APKTool. https://code.google.com/p/android-apktool/.

[11] Apple App Store. http://itunes.apple.com/.

[12] I. Asrar. Animal rights protesters use mobile means for their message. http://www.symantec.com/connect/blogs/animal-rights-protesters-use-mobile-means-their-message, 2011.

[13] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.

[14] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[15] C. A. Castillo. Android malware past, present, and future. *McAfee),[online]*, 2010.

[16] CyanogenMod. http://www.cyanogenmod.org/.

[17] CyanogenMod Statistics. http://stats.cyanogenmod.org/.

[18] D. Damopoulos, G. Kambourakis, and S. Gritzalis. isam: an iphone stealth airborne malware. In *Future Challenges in Security and Privacy for Academia and Industry*, pages 17–28. Springer, 2011.

[19] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[20] B. Elgin. Google buys android for its mobile arsenal. `http://www.businessweek.com/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal`, 2005.

[21] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6, 2010.

[22] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

[23] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, page 3. ACM, 2012.

[24] D. Fisher. Android 4.1 jelly bean includes full aslr implementation. `http://threatpost.com/en_us/blogs/android-41-jelly-bean-includes-full-aslr-implementation-071612`, 2012.

[25] Gartner. Gartner says worldwide mobile phone sales declined 1.7 percent in 2012. `http://www.gartner.com/newsroom/id/2335616`, 2013.

[26] GingerMaster: First Android Malware Utilizing a Root Exploit on Android 2.3. `http://www.csc.ncsu.edu/faculty/jiang/GingerMaster/`.

[27] Good LiveWallpaper. `https://play.google.com/store/apps/developer?id=Good+LiveWallpaper`, accessed April 16, 2013.

[28] Hackmageddon. One year of android malware (full list). `http://hackmageddon.com/2011/08/11/one-year-of-android-malware-full-list/`, 2011.

[29] N. Hardy. The confused deputy:(or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[30] Heart Live Wallpaper. `https://play.google.com/store/apps/details?id=com.livewallpaper.livewallpaper.jjhearts`, accessed April 16, 2013.

[31] G. Hua, Y. Fu, M. Turk, M. Pollefeys, and Z. Zhang. Introduction to the special issue on mobile vision. *International Journal of Computer Vision*, 96(3):277–279, 2012.

[32] Instagram - Google Play Store. `https://play.google.com/store/apps/details?id=com.instagram.android`.

[33] Launcher Pro. `https://play.google.com/store/apps/details?id=com.fede.launcher`, accessed April 16, 2013.

[34] H. Lockheimer. Android and security. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`, 2012.

[35] K. Mahaffey. Security alert: Droiddream malware found in official android market. `https://blog.lookout.com/blog/2011/03/01/security-alert-malware-found-in-official-android-market-droiddream/`, 2011.

[36] S. Mansfield-Devine. Android malware and mitigations. *Network Security*, 2012(11):12–20, 2012.

[37] D. Maslennikov. Find and call: Leak and spam. `http://www.securelist.com/en/blog/208193641/Find_and_Call_Leak_and_Spam`, 2012.

[38] E. McCallister. *Guide to protecting the confidentiality of personally identifiable information*. DIANE Publishing, 2010.

[39] M. Meeker. 2012 kpcb internet trends year-end update. KPCB, 2010.

[40] T. Nash. An undirected attack against critical infrastructure. Technical report, Technical Report, US-CERT Control Systems Security Center, 2005.

[41] New In Android 4.1. http://developer.android.com/about/versions/android-4.1.html#Permissions.

[42] NQ Mobile. Nq mobile 2013 security report. http://www.nq.com/2012_NQ_Mobile_Security_Report.pdf, 2013.

[43] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically rich application-centric security in android. *Security and Communication Networks*, 5(6):658–673, 2012.

[44] Open Handset Alliance. Industry leaders announce open platform for mobile devices. http://www.openhandsetalliance.com/press_110507.html, 2007.

[45] Open Handset Alliance - Android Overview. http://www.openhandsetalliance.com/android_overview.html.

[46] C. Papathanasiou and N. J. Percoco. This is not the droid you're looking for... *DEF CON*, 18, 2010.

[47] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 347–356. ACM, 2010.

[48] F. Reis Santos Mata et al. Leadership in the mobile smartphone market. 2012.

[49] G. Russello, B. Crispo, E. Fernandes, and Y. Zhauniarovich. Yaase: Yet another android security extension. In *Privacy, security, risk and trust (passat), 2011 ieee third international conference on and 2011 ieee third international conference on social computing (socialcom)*, pages 1033–1040. IEEE, 2011.

[50] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS)*, pages 17–33, 2011.

[51] Seabed Live Wallpaper. https://play.google.com/store/apps/details?id=wave.live.jwallpapers.

[52] Sebastian. yummy yummy, gingerbreak! http://c-skills.blogspot.com/2011/04/yummy-yummy-gingerbreak.html, 2011.

[53] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[54] Spreitzenbarth. Current android malware. http://forensics.spreitzenbarth.de/android-malware/, 2013.

[55] A. Thampi. Path uploads your entire iphone address book to its servers. http://mclov.in/2012/02/08/path-uploads-your-entire-address-book-to-their-servers.html, 2012.

[56] Trend Micro - Mobile App Reputation. http://m.trendmicro.com/mobileappreputation.

[57] W. Tsai. Malware for ios? not really. http://blog.trendmicro.com/trendlabs-security-intelligence/malware-for-ios-not-really/, 2012.

[58] J. Van Grove. Your address book is mine: Many iphone apps take your data. http://venturebeat.com/2012/02/14/iphone-address-book/, 2012.

[59] A. VARKOKOV. Android malware in the open marketplace. https://blog.avast.com/2011/12/13/android-malware-in-the-open-marketplace/, 2011.

[60] A. VARKOKOV. Dont think alternative markets save your money. https://blog.avast.com/2012/05/14/dont-think-alternative-markets-save-your-money/, 2012.

[61] Wikipedia Mobile OS Marketshare. http://en.wikipedia.org/wiki/File:World_Wide_Smartphone_Sales_Share.png accessed April 16, 2013.

[62] Wikipedia Mobile OS Sales. http://en.wikipedia.org/wiki/File:World_Wide_Smartphone_Sales.png accessed April 16, 2013.

[63] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu. pbmds: a behavior-based malware detection system for cellphone devices. In *Proceedings of the third ACM conference on Wireless network security*, pages 37–48. ACM, 2010.

[64] Z. Xu, K. Bai, and S. Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks*, pages 113–124. ACM, 2012.

[65] S. Yin. Android malware found in fake 'angry birds,' 'cut the rope,' and more. http://securitywatch.pcmag.com/none/291627-android-malware-found-in-fake-angry-birds-cut-the-rope-and-more, 2011.

[66] D. B. Yoffie and R. Kim. *Apple Inc. in 2010*. President and Fellows of Harvard College, 2010.

[67] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[68] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *Trust and Trustworthy Computing*, pages 93–107. Springer, 2011.