SYSTEM AND FRAMEWORK FOR IDENTIFICATION OF INFO MALWARE ON THE
ANDROID OPERATING SYSTEM

BY

ROBERT 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

The tremendous rise of mobile computing has led to an equally strong rise in mobile malware. We address this issue with several novel concepts, permission fingerprints and user-app agreements. We also present a groundbreaking and substantial new dataset of the state of Android Apps, and provide unique results gathered from it. Finally, we present AndroMEDA, an Android Security Extension which helps address many of the shortcomings to existing techniques of malware identification currently available.

This *seriously* needs to be extended.

To my

Parents, Lisa and Mark

Acknowledgments

I would like to thank my advisor Roy H. Campbell for his mentorship and guidance, as well as Alejandro Gutierrez for putting up with me. ?Yeah this needs to be rewritten.?

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Thesis Statement: Malware on Mobile Operating Systems, especially Android, is better understood when not just analyzing the capabilities of an application, but the expectations the user has as to how it utilizes those capabilities as well.

The rise of smartphones in the last 6 years has been nothing short of meteoric. Since the launch of the Apple iPhone in 2007, over ?1 billion or so? smartphones have been sold, from over ?100 or so?(Author, Year) manufacturers. 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 (Author, Year). Indeed, smartphones continue to become the focal point of almost all personal computing, and consequently the operating systems they run become more important and powerful.

Mobile operating systems (mobile OSs), like the PC operating systems of the 1990s, have a few major players that wield the most influence. The two largest operating systems in the mobile area are Android and iOS. Apple's iOS, made exclusively for the Apple iPhone and iPad, currently runs on over 15%(Author, Year) 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 a 75% marketshare. Some of the less popular, but still significant mobile operating systems are Windows Phone, with 2%, and Blackberry, with 4%.

?Introduce iOS and Android!? Android was started in 2003 by Andy Rubin and Android, Inc, previously the makers of the T-Mobile Sidekick. In 2005 they were acquired by Google Inc, and work continued in secret until its first release in 2008 with the HTC Dream.

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".

This security model has dramatically changed the nature of mobile software, and in turn, mobile malware. By forcing malware to fit inside of this security sandbox, malware authors must choose to either break out of the box, or work inside of it. This constriction has blurred the definition of mobile malware, and ultimately has profound implications for the user of the mobile device. We will examine these implications, and propose new tools and methods to help understand the nuances of modern malware, as well as provide a framework of tools to detect them.

Chapter 2

Background & Motivation

To understand the nature of modern mobile malware, we first examine the context. The two main models, that of iOS and Android, are compared and contrasted here.

2.1 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.

2.1.1 Personally Identifiable Information

Personally Identifiable Information (PII), as defined by NIST, 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" [22]. Mobile devices, having replaced cameras, cell phones, GPS devices, and PCs, 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

---

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

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.

2.1.2 The "App Stores" and Controlled Distribution

The final major difference between mobile OSs and PC OSs is the distribution of code. No mobile OS allows code to be ran outside of the sandbox, and all of them require the user to agree 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.

The first major distribution platform for mobile apps was Apple's App Store[8]. It's 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 these times of platforms, 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, apps still must request permission from the user to perform certain tasks.

2.1.3 Android Permissions

Android's distribution platform takes a different approach, and at it's core is also Android's core security model: The Permission System. 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 (formally 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. 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.

This core concept behind Android security - Permissions as a static contract of capabilities, is the classic example of a rule-based security model. ?expand on this?. However, in this paper we propose an alternate means of conceptualizing security, which focuses on the interrelationship between the user's perception of the app, and the app's actual behavior. We call this the "user-app agreement", and will elaborate on it more shortly.

2.2 Mobile Malware

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. Over ?201 or so? known OS-level exploits exist for iOS, where only ?5 or so? have been publicly released for Android. On the contrary, no side-loading is possible for iOS, and there have been very few instances of malware sneaking past Apple's App Store review process, although it has happened[2]. Android's malware situation is very much a product of the sideloading and lack of review process found in GPStore. Of all the mobile malware found for iOS and Android, over ?some percent? used no system-exploits at all, ?some percent? on the main app distribution platform.

Malware, as defined by the US-DHS, 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" [23]. On mobile devices, the dominant goal of malware is to gather information that leads to loss of privacy, found in over ?some large percent? of known mobile malware.

This trend, of malware that possesses no system exploits, but gathers information that leads to loss of privacy, 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.

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 with the most comprehensive android app database available. Finally, we propose several novel improvements to the Android security architecture, called AndroMEDA, aimed at building off of our conceptual work.

---

[2]In July 2012, SecureList noticed an iOS app that uploaded all of the user's contacts to a remote location without their consent. [21]

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 strips 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.

The four 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 it's own inter-app communication built off of it's 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.

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, 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 3.1.

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 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* 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 it's 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 1 new permission has been added, where previous apps always had access.

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 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.

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

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 a personal favorite: *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 it's behavior.

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 are signed with the system key, it can be assumed that they have the same capabilities. These capabilities are permanent as well, as permissions 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). This has caused two main vectors of attack: finding exploits to remotely install software, or masquerade as benign apps, and pass through trusted channels. On Android, this last technique is especially popular.

After masquerading as a benign app, malware on Android has three main techniques: privilege escalation attacks, monetary service attacks, and PII stealing attacks. The first one, privilege escalation attacks, takes many 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*.

4.1 Privilege Escalation Attack

4.1.1 Rooting

Rooting is the act of acquiring root privileges on an OS. Typically mobile OSs do not provide the user, or apps, with root permissions, 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*[19]. 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[11] - 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*[16].

GingerMaster 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. All previous exploits were patched in Android 2.3, 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 and patching system services. No known rootkits exist - malicious or not - for Android 4.0 and up - 54% of active Android devices.

4.1.2 Confused Deputy Attack

The second main vector for privilege escalation attacks is the Confused Deputy attack[17]. 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[9] address this by extending Android to analyze inter-app communication and detect this kind of attack. Despite all of these concerns, it's worth noting that no known malware has been used to exploit this.

4.2   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 sent messages, or calls. These attacks have been prevalent in the Android market for quite a while, but recent versions of android - after 4.2 Jelly Bean - have taken the step of warning the user before premium SMSs are sent. This attack vector, while common, will see a reduction in the future thanks to these measures.

?Expand on this more?

4.3   Private Info Attack

The last malware technique is the most significant, and represents the largest departure from typical malware. Apps that steal PII. 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. However, this is the biggest departure from typical malware. To the system, no unusual operations are performed, and no exploits are ran. The qualification for malware in this category thus lies in the "use" vs "misuse" of PII. Often times, this line is blurred.

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[29]. 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"[31]. 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[3], 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.

4.4   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 consenting and trusting specific actions an app may take.

An example of UAA can be seen in the expected behavior of two different hypothetical apps and a hypothetical user. One is a large Social Networking app, which requests permission to access internet, send SMS messages and read the contacts database. The other is a little known developer's game, which also requests permission to access internet, send SMS messages and read contacts. These apps have the same Permission Fingerprint, but the exact behavior of how they access these capabilities may or may not violate the UAA. 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 will violate the UAA (building off the example of Path).

In the case of the game, 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 fall within the UAA. However, if it sends SMS messages to your friends telling them to download the game, this would quickly violate the UAA.

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 PII malware.

4.5 Research Malware

In the realm of research malware, several prominent examples further demonstrate the vague line between use and misuse of PII, and the User-App Agreement. The most notable one is SoundComber[27] 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 research malware on Android is TapLogger[34]. 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.

4.6 Malware Summary

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 three main categories of attacks are privilege escalation attacks, monetary service attacks, and PII attacks. 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, PII attacks, 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 UAA.

Chapter 5

Related Works

?THIS WILL/MAY BE GREATLY EXPANDED UPON LATER?

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[24] built off of the Android Permission Framework by implementing runtime policies the user could define per-app. Later on, projects like TISSA[36] 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[14].

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[26] 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[15]. 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[28], developed in 2010, pBMDS[33] and Crowdroid[10] all attempt to classify malware based upon it's 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[18]. Introduced in 2011, Google Bouncer is a system that runs malware in a highly-observed sandbox, and watches for suspicious behavior. Since it's release, it's been the subject to quite a bit of criticism[20], with researchers finding over 20 ways to circumvent it.

Along similar lines, TrendMicro provides it's solution, App Reputation[30]. It runs apps in a "cloud" sandbox, watching for connections to suspicious websites, as well as other monitoring. A research project, Paranoid Android[25], 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.

5.3   Shortcomings

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.

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 more generally the UAA, is an extremely important part of malware detection that's missing from modern android security frameworks.

17

Chapter 6

Architecture

6.1 Motivation

AndroMEDA is designed to complement existing Android security extensions - little attempt was made to replicate the efforts of others. The goal in building AndroMEDA was the minimal amount of changes to the Android system, to extract sufficient 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 it's 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 on Android. 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[12]. CyanogenMod is a 3rd party open source Android distribution created by volunteers. It provides firmware for a wide variety of Android devices, and it's 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[13].

6.2 AndroMEDA Architecture

Android, as an open source project, is hosted at `source.android.com`[7], as a series of *git* repositories managed by a meta-script called *repo*. It's source tree is organized by project type, "frameworks", "external" and "libcore" 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".
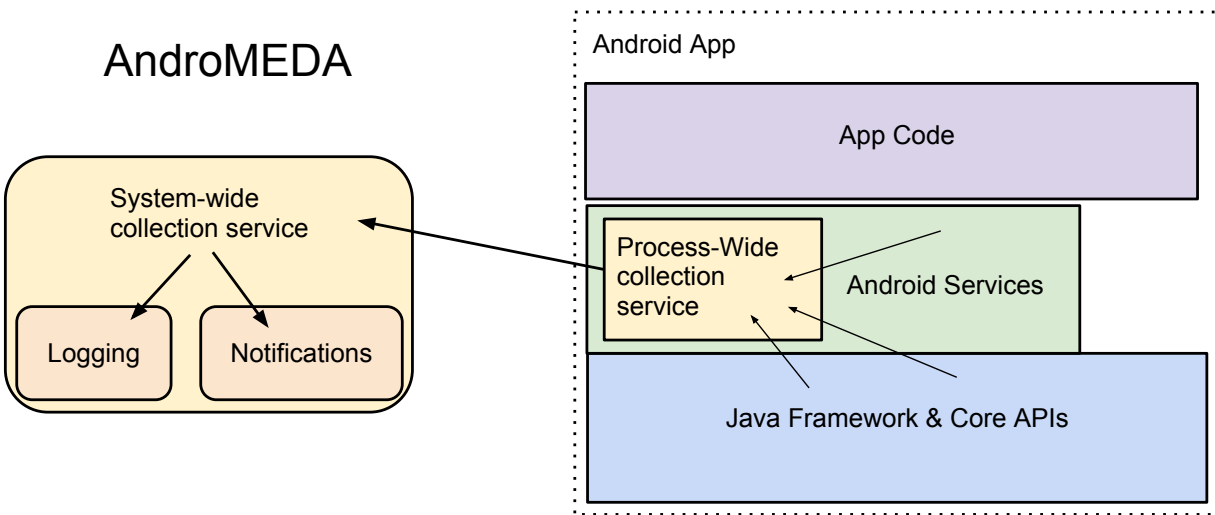
Figure 6.1: AndroMEDA Architecture Overview
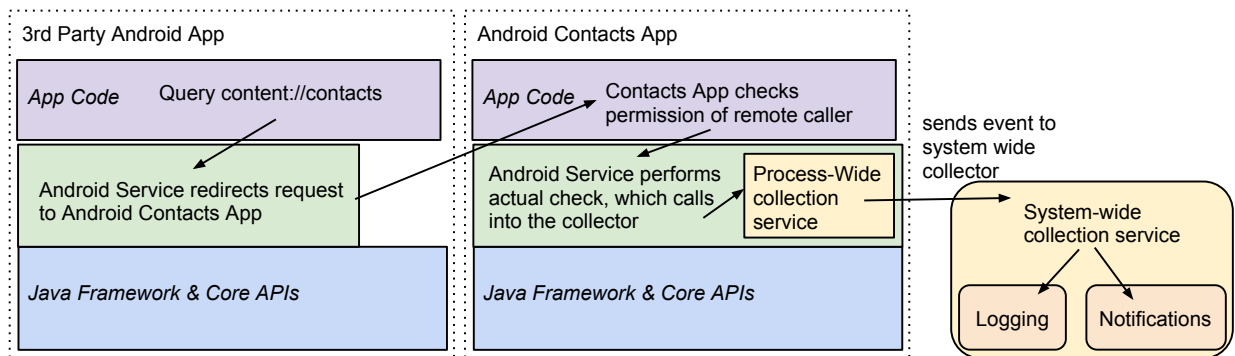
Figure 6.2: Example code path of an app requesting data from the Contacts app

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.

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. For the more difficult APIs, a
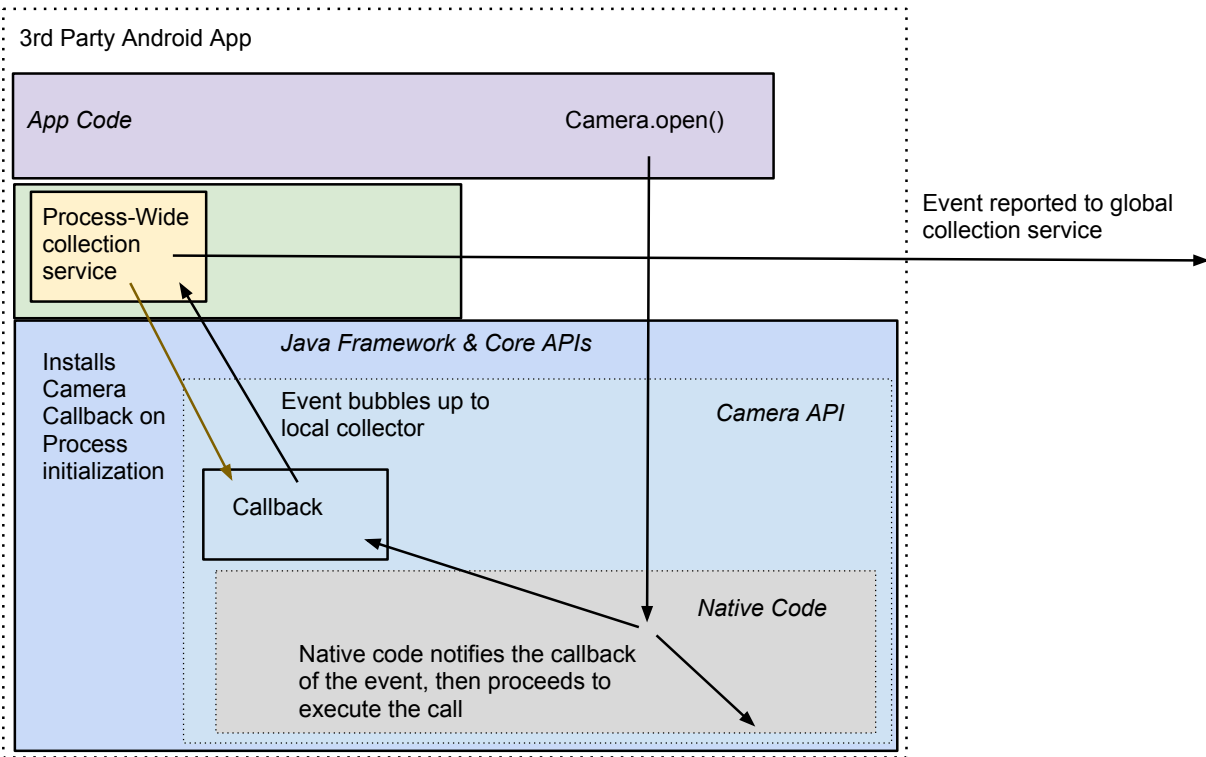
Figure 6.3: Example code path of instrumenting the Camera API

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. A quick null-check and then call are 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 it's 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 3, Permissions are the single security measure that defends an Android user from malicious software. We hope to 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 Market Dataset

To do a broad scale study of Android permissions, we introduce a novel dataset: ?AndroidMarketDB?. ?AndroidMarketDB? contains metadata from ?80%? of all known Android Apps in the GPStore, see Table 7.1. This metadata, described in 7.2, is rich in contextual information about an app. For a period of 1 month in May-June 2012, the GPStore was scanned twice a day. This dataset provides detailed metadata for all areas of the Android market, not just the top percent. Despite the large amount of metadata available, we will focus on the Permissions, leaving most other fields for Future Work9.1.

7.2 Global Permission Analysis

We first look at the global trends of Permissions in the GPStore. Ideally, we find that access to sensitive PII, and dangerous operations, only is requested by a handful of apps that truly need them. This 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 expect to find a system that falls short of these goals.

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

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

Table 7.1: Statistics from ?AndroidMarketDB?

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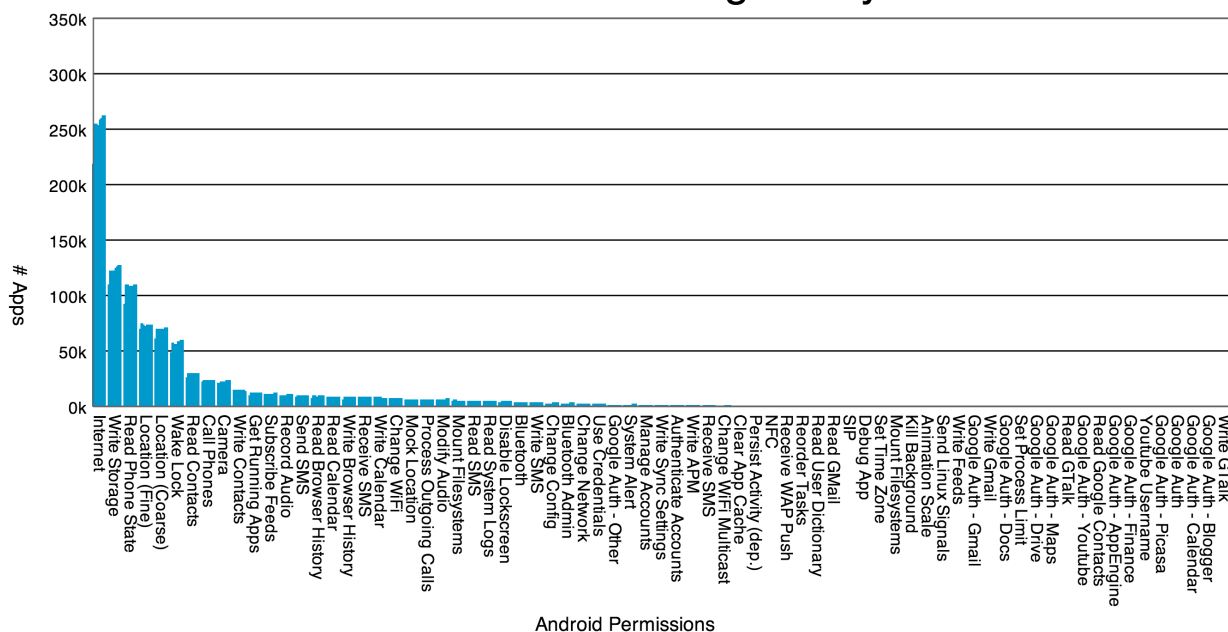
Figure 7.1: Permissions, sorted by how many apps request them in the entire GPStore dataset

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

23
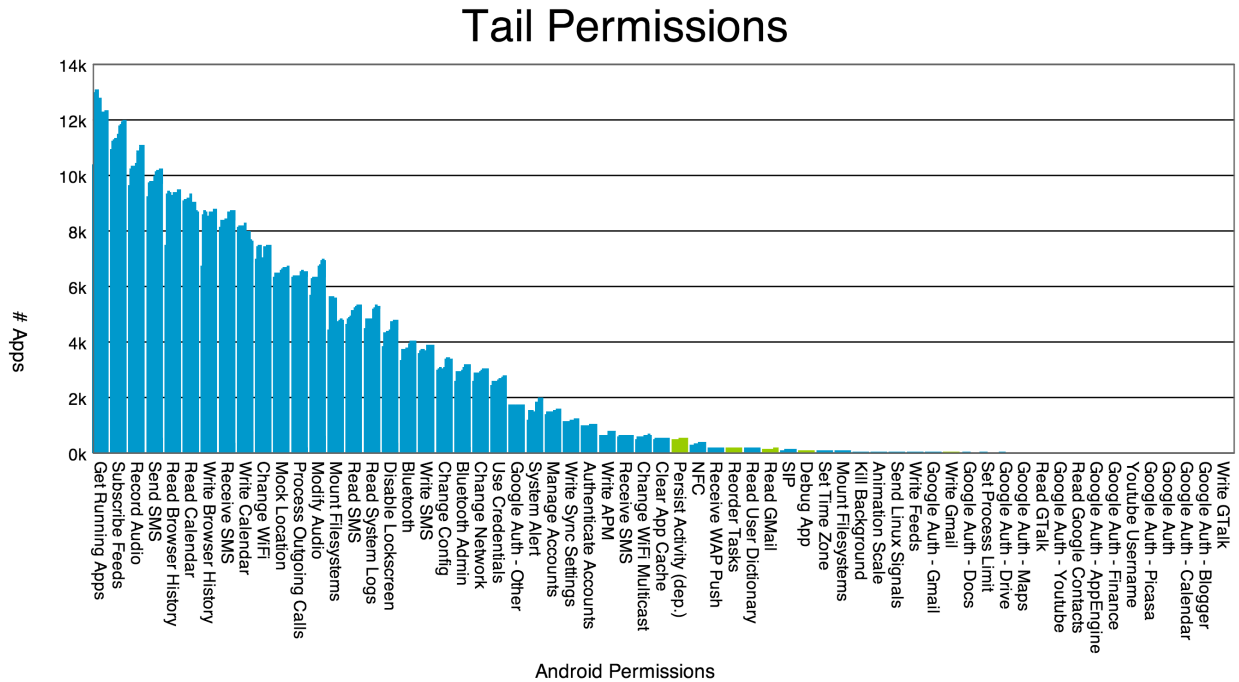
Tail Permissions

Figure 7.2: Less commonly used permissions in the GPStore

relatively small portion of the GPStore, still occupy a substantial portion. Indeed, with every dangerous permission, many legitimate use cases can be established, see Figure 7.3 for examples.

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

24

7.2.1 Per 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 ?AndroidMarketDB? 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 predominantly 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 it's 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*; it's 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.

?STUDY THESE THINGS AND REPORT WHAT THE HECK IS UP?

Overall, we've noted a number of times when patterns in permissions in the GPStore break our expectations of them - potentially violating them User-App Agreement. These could be detected simply from the Permission Fingerprint. We have shown a fairly strong correlation with major PII permissions and Android categories, but demonstrated many cases where they do not line up, both maliciously and with legitimate use cases.

7.3 Malware Dataset

Next, we compare a malware dataset with ?AndroidMarketDB?, in the hopes of finding them as outliers. Ideally, all malicious behavior would show up in the permission fingerprint of an app, so therefore it's unique set of capabilities would stand out in the Google Play Store. Demonstrating that expected behavior and Permission Fingerprint are correlated will be an important part of that. Ultimately, however, we expect this method to have it's shortcomings.

7.3.1 Android Malware Genome Project

The dataset we use - the largest academic set of it's kind, is the Android Malware Genome Project[35], from NCSU. Containing almost 1300 samples from 52 families of actual Android malware, it provides the ideal test dataset for Android. We first classify each malware family by its capabilities, according to data aggregated by Spreitzenbarth[2]

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

and Hackmageddon[1], 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 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 Malware 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, it's 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 systme 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[1]. We also observe that *Botnet* malware occasionally accesses system features like disabling the lock screen and reading system logs.

These obversvations are highly valuble, 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.

---

[1]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

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 ?AndroidMarketDB?

7.4 Popular Apps

As we have shown, all categories of malware but Info Stealing show profound differences in permission fingerprints: they are far more likely to access unusual permissions than normal Android apps. However, Info stealing apps did not show such anomalies, making them far easier to hide inside of benign apps - a technique malware writers already use[32]. Thus, if more popular apps have more access to PII, this vector poses a serious threat.

For starters, we note that the metadata in ?AndroidMarketDB? doesn't give precise download measurements, but rather broad ranges. We plot these in Table 7.5, noting the number of apps in each download range, and it's total percent of all downloads in the market. The top percentile, only 9 apps, accounts for an estimated[2] 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.

?Show the same permission histogram as before, but each row is a different download range?

Discuss it.

7.5 Market Summary

We have demonstrated several key points in this analysis. First, we demonstrated that Permission Fingerprints, and their aggregate histograms, 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

---

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

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-stealing 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-stealing malware to imitate their permission fingerprints.

7.6 Shortcomings of Permissions on Android

We have established the limits of Permission Fingerprints: Whereas they list capabilities, they are broad, and do not offer the insight necessary to identify malicious behavior. We have demonstrated that malicious behavior and acceptable behavior can come from the same Permissions. This is ultimately due to two main issues, Permissions do not address the context of use, nor what is done with the data after granting it. These two ideas, context and use, are some defining aspects of the User-App Agreement.
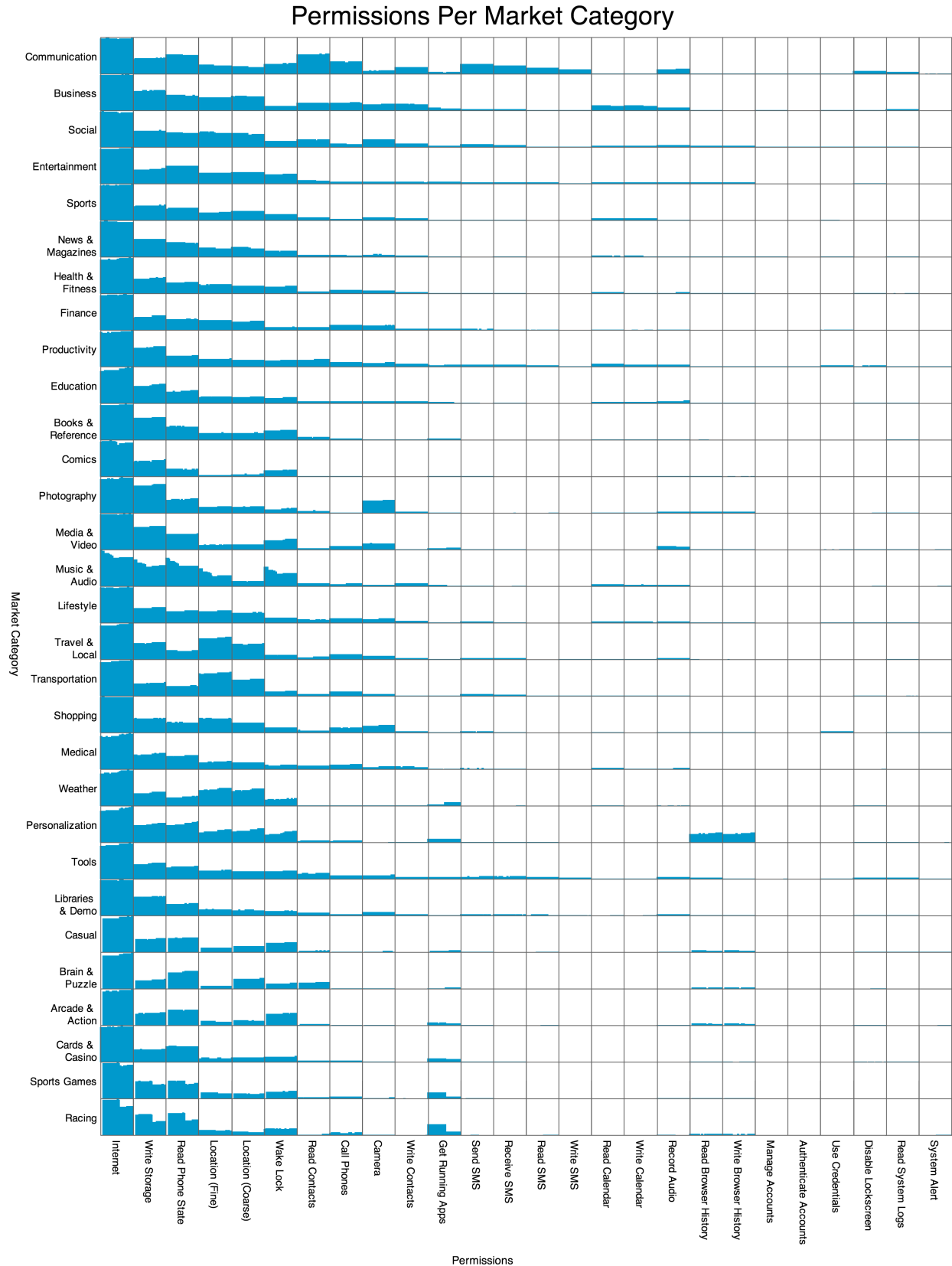
Figure 7.3: Permissions used, as a fraction of total in that category

Android Malware Genome Project

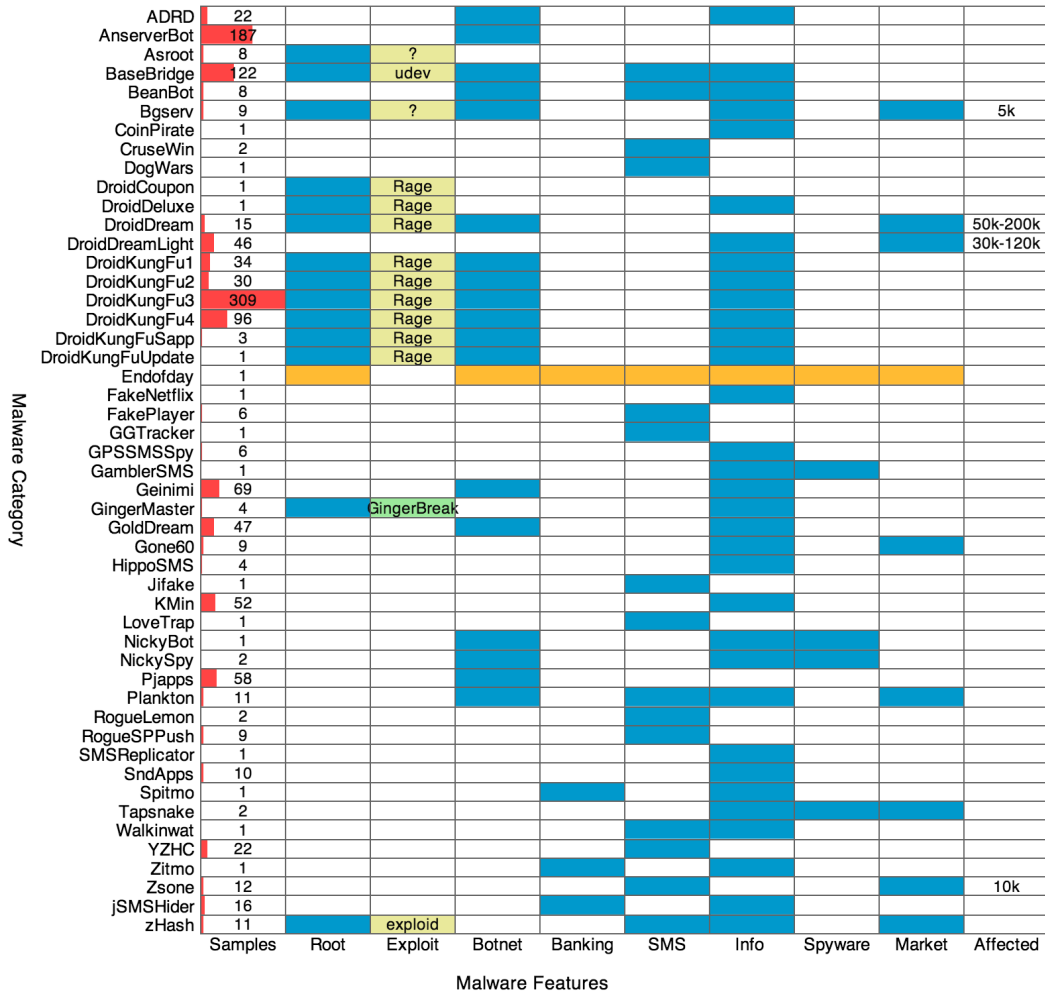| Malware Category	Samples	Root	Exploit	Botnet	Banking	SMS	Info	Spyware	Market	Affected
ADRD	22			●			●			
AnserverBot	187									
Asroot	8	●	?							
BaseBridge	122	●	udev	●		●	●			
BeanBot	8					●	●			
Bgserv	9	●	?	●			●		●	5k
CoinPirate	1									
CruseWin	2					●				
DogWars	1									
DroidCoupon	1	●	Rage							
DroidDeluxe	1	●	Rage				●			
DroidDream	15	●	Rage	●			●		●	50k-200k
DroidDreamLight	46						●		●	30k-120k
DroidKungFu1	34	●	Rage	●			●			
DroidKungFu2	30	●	Rage	●			●			
DroidKungFu3	309	●	Rage	●			●			
DroidKungFu4	96	●	Rage				●			
DroidKungFuSapp	3	●	Rage				●			
DroidKungFuUpdate	1	●	Rage				●			
Endofday	1	●	●	●	●	●	●	●		
FakeNetflix	1						●			
FakePlayer	6					●				
GGTracker	1					●				
GPSSMSSpy	6						●	●		
GamblerSMS	1						●	●		
Geinimi	69			●			●			
GingerMaster	4	●	GingerBreak				●			
GoldDream	47			●			●			
Gone60	9						●		●	
HippoSMS	4					●	●			
Jifake	1					●				
KMin	52						●			
LoveTrap	1					●	●			
NickyBot	1			●			●			
NickySpy	2			●			●	●		
Pjapps	58			●		●	●			
Plankton	11			●			●		●	
RogueLemon	2					●	●			
RogueSPPush	9					●	●			
SMSReplicator	1						●	●		
SndApps	10						●	●		
Spitmo	1				●		●			
Tapsnake	2						●	●	●	
Walkinwat	1					●	●			
YZHC	22					●	●			
Zitmo	1				●		●			
Zsone	12					●	●		●	10k
jSMSHider	16				●	●	●			
zHash	11	●	exploid	●					●	

Malware Features

Figure 7.4: A summary of the malware families found in the Android Malware Genome Project
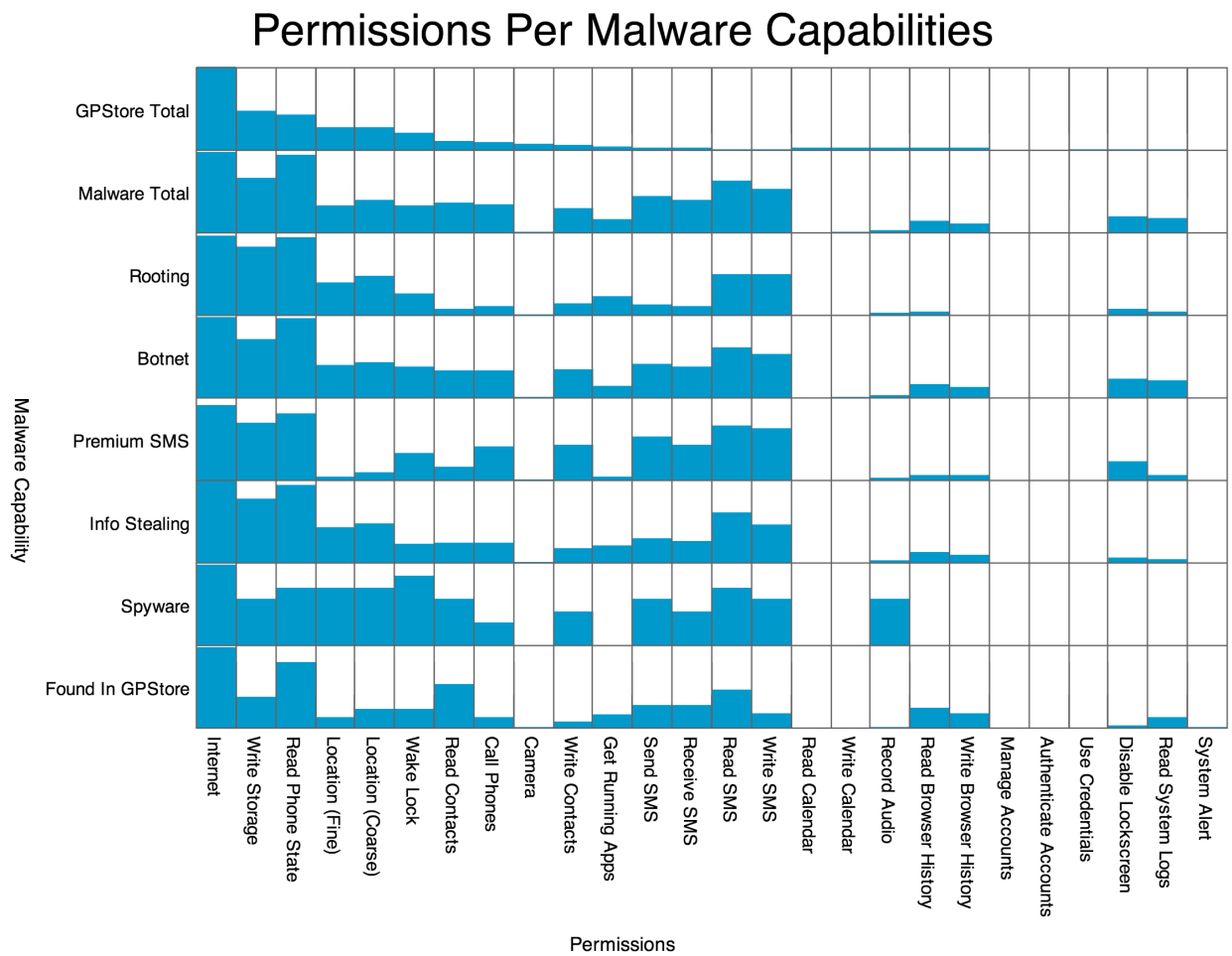
Figure 7.5: Permission Fingerprints of malware with different capabilties, compared to the GPStore total

Chapter 8

Framework 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 reguardless of any context - at any time, and reguardless of user consent. The data, after being requested, ultimately can be manipulated and trasmitted to any party without user consent.

Projects like TaintDroid[14] have begun to address the flow of personal data, which aids in the user understanding the use of data. Ultimately, however, much more can be done to address both context and use, which AndroMEDA attempts to address. By instrumenting API calls, AndroMEDA can both inspect context and to some extent use of personal data and other system actions, and by visualizing this information to the user, provides them with the ability to evaulate whether the User-App Agreement has been broken.

8.1   Test Introduction

To test the effectiveness of AndroMEDA at detecting malicious behavior, we begin with testing apps from the Android Malware Genome Project[35]. Unfortunately, we found only 31 of the nearly 1300 samples were designed for Android 2.3 and above, and not a single one targeted Android 4.0 or above. Furthermore, we found many of the samples simply did not run anymore, due to deprecated APIs and other poor coding techniques.

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 Stealing Apps. We introduce a novel sample of info stealing malware, embedded inside of existing software - designed to silently steal PII and upload it to a remote location. We build off of concepts like SoundComber[27]

Chapter 9

Conclusion and Future Work

9.1   Future Work

References

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

[2] Current android malware. `http://forensics.spreitzenbarth.de/android-malware/`, 2013.

[3] 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`.

[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 Open Source Project. `http://source.android.com/`.

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

[9] 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.

[10] 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.

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

[12] CyanogenMod. `http://www.cyanogenmod.org/`.

[13] CyanogenMod Statistics. `http://stats.cyanogenmod.org/`.

[14] 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.

[15] 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.

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

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

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

[19] 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.

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

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

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

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

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

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

[33] 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.

[34] 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.

[35] 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.

[36] 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.