Cool. A Paper.

ABSTRACT

This is abstract

1. INTRODUCTION

Figure 1: Worldwide Market Share of various mobile OSs - from [48] and [20]

Mobile OSs, like the PC operating systems of the 1990s, have a few major players that wield the most influence, as seen in Figure 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, runs on over 20%[20] of all smartphones globally, as of the end of 2012. 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%[20] marketshare. Some of the less popular though still significant mobile operating systems are Windows Phone, with 3%, and Blackberry, with 3.5%[20] .

1.1 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"[53]. However, "the Apple iPhone was a huge success. Considered by Time magazine the invention of the year 2007 ... it completely changed the mobile phones industry dynamics"[37]. Apple's iPhone and iOS were novel due to their touch friendly and intuitive OS, as well as their digital distribution platform, the App Store[53]. In 2012, Apple sold over 130 million iOS devices[20].

1.2 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[15]. "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"[34]. Since its initial release in 2007[33], Android has sky-rocketed 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[20] — more than Apple had the entire year, as seen in Figure 2.

Figure 2: Worldwide Sales of various mobile OSs - from [49] and [20]

1.3 Goals of Mobile OSs

All these mobile OSs share many common goals and challenges. The diversity of hardware that smartphones were designed to replace, along with 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 include:

- Small memory footprint, battery-conscious, and other resource restrictions
- Access to a wide variety of personally identifiable information (PII)
- Access to 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*.

1.4 The *App* and Sandboxing

In the mobile world, *Apps* are isolated and sandboxed programs, generally designed with a singular purpose. They lack dependencies and generally less privileged than 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 nonexistent (with some exceptions[1]), along with many other restrictions. On Android and other mobile 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.

Various reasons exist for the tight sandboxing of mobile apps. Power and resource consumption are certainly a factor — mobile OSs generally reserve the right to terminate apps if they attempt to allocate too much memory. Controlling access to hardware also helps with 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.

1.4.1 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 individual's identity... and (2) any other information that is linked or linkable to an individual" [29]. Mobile devices — having blended cameras, cell phones, GPS devices, and PCs into one device — have an extremely diverse amount of PII, including phone numbers, contacts, location history, bank account numbers and pictures. For many of these datasets, mobile OSs actually organize them into databases with the intention of allowing third 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 third party software's access to PII.

1.4.2 Digital Distribution Platform

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

1.4.3 Apple's App Store

The first major digital distribution platform for mobile apps was Apple's App Store[6]. 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 is released onto the app store, at which point 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 3.7.1), apps must still request permission from the user to perform certain tasks.

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

1.4.4 Android Permissions

Android's distribution platform takes a different approach. At its core is Android's security model: The Permission System (see section **??**). When Android Apps are packaged, they declare what capabilities they will use and the user reviews them before installation. If the user approves the app, it may use the requested capabilities at any time: 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, in which the developer pays a smaller fee and apps do not go through a formal approval process. After an app's submission, it is immediately released to the public for users to download and run. Android assumes 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 is given (see Figure 3). In fact, Android even allows the device to accept apps from third party sources — a practice known as *sideloading* — although it is disabled by default. This has spawned a large number of third party app sources, all of which rely on the Permission system for user protection.

Figure 3: A sample page on the Google Play Store, see 1

1.5 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" [30]. 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 [13]. Alternately, the first Android exploit was not discussed until 2010 by security researchers Papathanasiou and Percoco [35], and was not seen in use until early 2011[10]. On the contrary, no sideloading 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]). With 95% of all mobile malware[31], An-

[2]In July 2012, SecureList noticed an iOS app that uploaded

droid's malware situation is a product of the sideloading and lack of review process found in GPStore[31].

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[31]. The trend of malware possessing no system exploits, but gathering information that leads to loss of privacy — 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 upon installation 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 without exploits. This all works against the user.

This paper attempts 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 expanding on of our conceptual work.

2. RELATED WORKS

2.1 Android Extensions

AndroMEDA is far from the first to attempt to address the issue of malware on Android. As early as 2009, frameworks such as SAINT[32] built off of the Android Permission Framework by implementing runtime policies the user could define per app. Later on, projects like TISSA[55] built off of SAINT by implementing varying levels of obfuscated data. When an app running on TISSA requests access to the contacts database, TISSA can either provide the app with the full database, a limited portion, an anonymized portion, or no information. Not all PII was covered in their framework, 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, unlike TaintDroid[16].

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[38] 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 by many accounts is somewhat negligible, certainly plays a role in a decision to incorporate such security extensions.

all of the user's contacts to a remote location without their consent[28], but others argued this was not as devious as made out to be[45]

Second, the complex rulesets required build off of the Permission Framework, adding additional rules the user may configure. However, a study by Berkeley suggests only 17% of users paid attention to permissions at install time, and only 3% correctly remembered them later[18]. Clearly, tasking the user with more work is the wrong 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[41] (developed in 2010), pBMDS[50], and Crowdroid[9] all attempt to classify malware based upon its interaction with the user. However, none actually ask immediate input from the user — a fundamental flaw that limits the ability to adapt to the user's specific UAAs.

2.2 Android Sandboxes

On the other side of malware detection is *automated* malware detection. The major project in this regard is Google's Bouncer[25]. 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 has been highly criticized[27], with researchers finding over twenty ways to circumvent it.

Similarly, TrendMicro provides its solution, App Reputation[44]. It runs apps in a "cloud" sandbox, watching for connections to suspicious websites, as well as other monitoring. A research project, Paranoid Android[36], 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.

Most all of the frameworks listed above, however, especially the sandbox tools, assume a clear ruleset to be established classifying malicious behavior versus benign behavior. They require a clear line to establish what constitutes trusted behavior versus untrusted. Unfortunately, this counters the concept of the UAA, in which every individual action has a complex set of rules that result in an acceptable behavior versus 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 seem malicious to some users, and perfectly normal to others.

2.3 Android Surveys

2.4 Contribution

WE DUN STUFF.

3. CONCEPT

3.1 Mobile Threat Model

Apps can steal PII undetected. Like Soundcomber.

Malware on mobile devices has seen a departure from past exploits. The wealth of Personally Identifiable Information easily available on mobile OSs increasingly makes them the focus of malicious software. In addition, the tight sandboxing constraints often forces 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 can not be run by the user unless it is in it is form and comes through a trusted channel (unless of course that security feature has been disabled). Once on the device, malware has several main methods of attack.

3.2 Installation

The three primary ways in which malware can be installed on an Android device are *Repackaging*, *Update Attack*, and *Drive-by Download*[54]. The first two are designed to sneak malware into the Google Play Store or other third 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 6.2). *Update Attacks* typically build off of *Repackaging*, but do not acquire malicious code until later, making static detection difficult[54]. The last, *Drive-By Downloads*, is often tied with *Repackaging*, but is not presented in an official app channel. Instead, it is downloaded when the user visits a webpage or clicks a link[54] and Android prompts the user as to whether or not they would like to install the app. All three methods involve concealing the intent of the malware and passing as a legitimate app; they do not use browser exploits or system exploits to install the initial malware without the user's consent.

3.3 Malicious Actions

Once installed on the device, Android malware has several main methods of attack. Xuxian Jiang and Yajin Zhou[54], along with Spreitzenbarth[42], and Hackmageddon[22] define four major categories: *Privilege Escalation Attack*, *Remote Control Attack*, *Monetary Service Attacks*, and *Privacy Info Theft.*

3.4 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.*

3.4.1 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 as all monitoring of apps relies on monitoring the sandbox — there is no way to track an app that escapes the sandbox. 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*[26]. 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 and/or permission. In March 2011, however, DroidDream was discovered. DroidDream used *RageAgainstTheCage* to silently install additional applications in the background, stealing PII and forming a botnet. By the time Google remotely removed it from the market, it had been downloaded an estimated 50,000 to 200,000 times[10], which was 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 exploits such as *RageAgainstTheCage*, *udev*, and another called *GingerMaster*[21].

3.4.2 Recent Android Rootkits

GingerMaster[40] is significant because it is the last known root exploit malware seen in use. Designed for Android 2.3.3, it can currently run on 45% of all Android devices in use, according to the official Android Dashboard[2]. In reality, however, of that 45%, almost all have been patched to fix the exploit[40]. All previous exploits were patched in Android 2.3[40], 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[19] and hardening system services[3]. No known rootkits exist — malicious or not — for Android 4.0 and up comprising 54% of active Android devices.

3.4.3 Confused Deputy Attack

The second main vector for privilege escalation attacks is the *confused deputy attack*[23]. In this scenario, services that guard sensitive operations are "tricked" into performing them. For example, if a Content Provider forgets to check a permission, or if a developer finds 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 by asking the web browser to open a URL. By including sensitive data in the URL, an app may still transmit sensitive data to a remote server without ever requesting the *INTERNET* permission. Projects like XManDroid[8] and Quire[14] address this by extending Android to analyze inter-app communication and detect this kind of attack.

3.5 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[54] found in 93% of Android malware — and is often used in conjunction with other attacks[42].

3.6 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 are only visible after the user checks their bill. These attacks have been prevalent in the Android market for quite a while, with NQ Mobile[31] listing it as one of the top three threats of 2012, and being found in 39 of 119 of the malware documented by Spreitzenbarth[42]. However, recent versions of Android (after 4.2 Jelly Bean[3]) have taken the step of warning the user before premium SMSs are sent.

A prime example of this is FakeInst[47], a repackaged version of Instagram[24] 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"[47]. In many cases, the premium SMS messages would end up being billed to the user for over $4 each[47], without ever alerting the user. Messages are often deleted by the app, removing the trace until the user gets their bill.

3.7 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 — including call logs, contacts, and photos — and send them to a remote server. This fits 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" versus "misuse" of PII; often times, this line is blurred.

3.7.1 Path on iOS

A large distinction of what constitutes as privacy malware to an individual stems from the user's expectations of how the app will use their PII. Consider the case of the Path iPhone app, which in February 2012 was discovered to be uploading the user's entire contact list to Path's servers, without any consent from the user[43]. It is 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"[46]. 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: 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 granted unrestricted access to the content[1]. This change, however, did not fully address situations like Path, in which is was less about the app simply having access to the data, and more about what the app actually did with the data behind the scenes. When these actions did not match up with user expectations, the app was treated as malware until the situation was cleared up by Path. The next day, Path issued an update that immediately explained to the user what they were going to do with the data and why.

It is 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, denying them the ability to decide for themselves if they are comfortable with them — supporting our motivation for AndroMEDA.

3.8 User-App Agreement

The Path 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, while apps that do follow this expectation 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, dealing instead with exactly when and how those actions are taken. Since this agreement is not formally defined, it is acquired through external trust in an app. This happens in various ways, through the description the app provides, 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.

3.8.1 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 "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 and break the trust of the user (expanding on the example of Path).

3.8.2 UAA Example - Social Game

The other example is a little known developer's game, which, like the social networking app, also requests permission to access internet, send SMS messages, and read contacts. 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. Furthermore, if it sends SMS messages to your friends telling them to download the game, this would breech the UAA and break the trust of the user.

In both examples, the apps have the exact same Permission Fingerprint, but vary widely in their expected behavior, and which actions are trusted and untrusted. This misuse of PII and other device capabilities fits with our definition of malware, and highlights the shortcomings of the Permissions framework: being unable to deal with the subtle differences between trusted behavior and untrusted behavior. Indeed, every user 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.

3.9 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[39]. 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, as well as 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[51]. TapLogger imitates a simple touch-based game, learning the vibration patterns of the device for each tap. By recording the vibration patterns in the background, it attempts to discover passwords and other sensitive keyboard events, all through a seemingly trusted sensor. TapLogger requests no Permissions; therefore its behavior is possible for any app.

In Chapter 6.2, we expand upon these examples to present an additional dataset of IncognitoWare – repackaged apps with malicious software added — designed for research purposes. We keep the Permission Fingerprints identical to the cloned app, making detection exceptionally difficult.

3.10 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 or semi-trusted channels to enter the device. Once on the device, the malware manifests itself as one of four main categories of attacks — privilege escalation attacks, remote-control attacks, monetary service attacks, and personal info theft. Of these categories, privilege escalation attacks and monetary service attacks are the easiest to protect against — and indeed Android has taken serious steps to mitigate these. The third time of attack, however, personal 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.

3.11 System Goals

Provide feedback to the user, allowing them to identitify misuse of PII. Of the attacks described above, the only ones we care about are PII attacks.

4. ANDROMEDA FRAMEWORK

4.1 Overview

Android Malware Evaluation Detection and Analysis (AndroMEDA) is designed to complement existing Android security extensions like TISSA[55], TaintDroid[16], YAASE[38], and XManDroid[8]. The goal in building AndroMEDA was making the minimal amount of changes to the Android system in order to extract sufficient amounts of information. Portability was important, as code that is easier to port can be more easily adopted. Along with being lightweight and fast, portability further forced the extension to be as simple as possible. The framework aimed to be as independent of hardware as possible, with the ability to work on any device: smartphone, tablet, TV, or future devices like Google Glass. This portability meant as few changes to the low-level drivers and kernel as possible.

Figure 4: AndroMEDA Architecture Overview

Functionally, 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, finding strategic opportunities to expose these events to the user, giving the user actions to perform in reaction, are key steps in mitigating malware — especially Info Theft Malware — on Android. Unfortunately, as discussed in Chapter **??**, Android does not log these sensitive events natively, thus motivating this framework. Ultimately, it is expected that a framework such as this would not eliminate malware on Android, but rather be part of a larger system.

As a base for our framework, we chose CyanogenMod[11], a third party open source Android distribution created by volunteers. It provides firmware for a wide variety of Android devices, and its user-base is comprised of 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[12].

4.2 Details

Android, as an open source project, is hosted at `source.android.com`[4], 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" are the top-level folders we focus on. "Frameworks" contains all of the Java, C++, C, assembly code, 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 4), is organized into two main parts: (1) a collection of hooks in the API and (2) 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.

4.2.1 System Extension

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 occurs during a Remote Procedure Call (RPC) with the process that owns the data; as such, the remote UID is sent along (as seen in Figure 5). Unfortunately, many permissions are not checked through

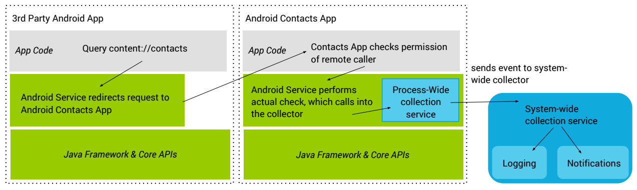

3rd Party Android App
App Code
Query content://contacts
Android Service redirects request to Android Contacts App
Java Framework & Core APIs
Android Contacts App
App Code
Contacts App checks permission of remote caller
Android Service performs actual check, which calls into the collector
Process-Wide collection service
Java Framework & Core APIs
sends event to system-wide collector
System-wide collection service
Logging
Notifications

Figure 5: Example code path of an app requesting data from the Contacts app

Figure 6: Example code path of instrumenting the Camera API

this method, but Android Permissions Demystified[17] 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 launching the process, our local collector populates these global variables with callbacks that marshal the data 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). Using this method, we are able to instrument any API call, getting more data than when the permission is simply 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, the hooks call into a local callback, which then calls the process-wide callback accordingly. An example can be seen in Figure 6.

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 are required.

The global service acts 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, as well as 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.

4.2.2 Companion App

We wrote a companion app 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 reports a log of events 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 the unavailability of local logs, the companion app can still query the web service, and view event histories that other users have published.

5. ANALYSIS

6. EVALUATION

With AndroMEDA, we attempt to build on top of the Android Permission system, to do a better job of 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, can ultimately be manipulated and transmitted to any party without user consent.

Projects like TaintDroid[16] have begun to address the flow of personal data, which aids in the user understanding the use of their data. Much more can be done, however, to address both context and use —which AndroMEDA does. By instrumenting API calls, AndroMEDA can both inspect context and use of personal data as well as other important system actions. By presenting this normally-hidden information to the user, AndroMEDA provides the user with a feedback loop to evaluate whether the User-App Agreement has been broken.

6.1 Existing Malware Datasets

To test the effectiveness of AndroMEDA at detecting malicious behavior, we begin with testing apps from the Android Malware Genome Project[54], the most comprehensive academic malware dataset. Unfortunately, as show in in Figure 7, we found only 31 (2.4%) of the nearly 1,300 samples were designed for Android 2.3 and above, when Android fixed many rootkits. Not a single one targeted Android 4.0 or above, when Android took steps to both harden security or Android 4.2, when Android to fix SMS related malware[3]. 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 four examples, one — *RogueSPPush* — sends premium SMS messages, but does not request *Send SMS*, rendering it ineffective. The three 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.

The remaining app, *DogWars*, makes a good testing app.

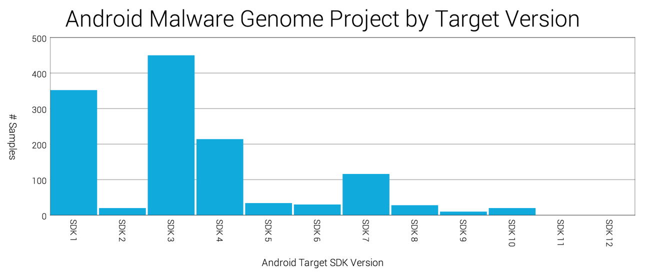
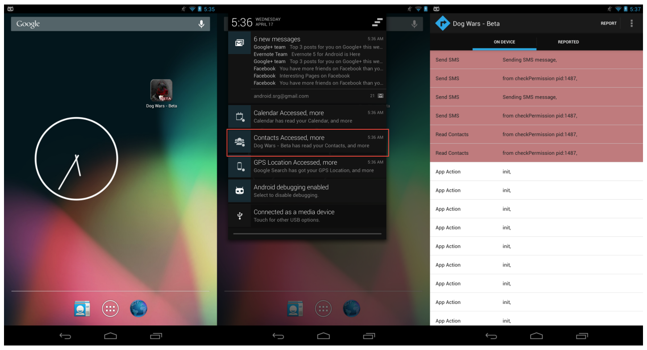


Figure 7: Malware from the Android Malware Genome Project by Android Version

Made as a "hacktivist" app in protest of an existing app[7], *DogWars* repackages a game with code to send SMS messages to all contacts on the next boot. The screenshots in Figure 8 show how AndroMEDA clearly identifies when *DogWars* accesses contacts and sends messages, allowing the user to identify it as malware.



6 new messages 5:36 AM
Google+ team Top 3 posts for you on Google+ this we...
Evernote Team Evernote 5 for Android is Here
Google+ team Top 3 posts for you on Google+ this we...
Facebook You have more friends on Facebook than yo...
Facebook Interesting Pages on Facebook
Facebook You have more friends on Facebook than yo...
android.srg@gmail.com 21

Calendar Accessed, more 5:36 AM
Calendar has read your Calendar, and more

Contacts Accessed, more 5:36 AM
Dog Wars - Beta has read your Contacts, and more

GPS Location Accessed, more 5:36 AM
Google Search has got your GPS Location, and more

Android debugging enabled
Select to disable debugging.

Connected as a media device
Touch for other USB options.

```
Send SMS        Sending SMS message,
Send SMS        from checkPermission pid:1487,
Send SMS        Sending SMS message,
Send SMS        from checkPermission pid:1487,
Read Contacts   from checkPermission pid:1487,
Read Contacts   from checkPermission pid:1487,
App Action      init,
App Action      init,
App Action      init,
App Action      init,
App Action      init,
App Action      init,
App Action      init,
App Action      init,
App Action      init,
```

Figure 8: 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. We have defined IncognitoWare as the concept of hiding Info Theft Malware inside of benign apps, designed to silently steal PII and perform other unwanted operations. 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.

6.2 IncognitoWare Dataset

IncognitoWare has recently become one of the most popular forms of mobile malware[31]. FakeInst, discussed in Chapter 3.6, was a repackaged version of Instagram[24] that sent premium SMS messages on start. While admittedly basic, more complex versions, such as FakeAngry[52] 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[31], 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*[5] 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 is suitable for deployment in the Google Play Store, or third party markets.

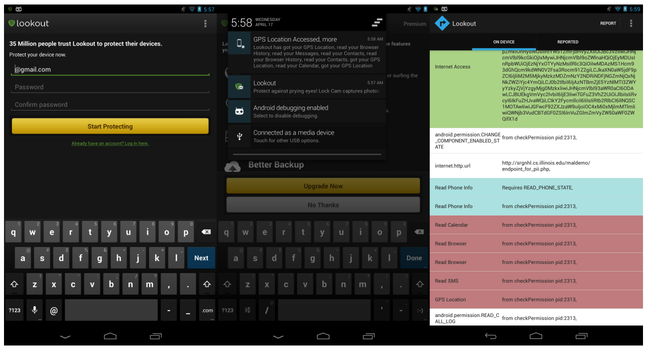
6.3 Info Theft IncognitoWare

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 it is not only 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, yet inconspicuous enough to be undetected, and legitimate enough to be unblocked.

AndroMEDA, however, can easily detect this example, seen in Figure 9. After the user logs in, AndroMEDA immediately alerts the user that the PII has been read. The user then reviews the data shown in Figure 11 and decides whether or not 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 10. It is worth noting that the logs in Figure 10 and 11 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.

6.4 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



Internet Access	pZmkiOnRydWUsIiRtYW51ZmFjdHVyZXIiOiJBc3VzIiwiJHNjcmVlbl9kcGkiOjIxMywiJHNjcmVlbl9oZWlnaHQiOjEyMDUsInRpbWUiOjEzNjYxOTYyNzMsIlRlc3QiOiIwMDAzMS1Hcm93dGhQcm9kdWN0V2Fsa3Rocm91Z2giLCJkaXN0aW5jdF9pZCI6IjliM2M5MjkyMzkzMDZmNzY2NDRiNDFjNGZmNjQxNjNkZWZiYjc4YmQiLCJ0b2tlbiI6IjAzNTBmZjE5YzNlMTI3ZWYyYzkyZjVjYzgyMjg0MzkxIiwiJHNjcmVlbl93aWR0aCI6ODAwLCJBUEkgVmVyc2lvbiI6IjE3IiwiTGFuZ3VhZ2UiOiJlbiIsIiRvcyI6IkFuZHJvaWQiLCIkY2FycmllciI6IiIsIiRtb2RlbCI6IlNQSC1MOTAwIiwiJGFwcF92ZXJzaW9uIjoiOC4xMi0xMjImMTImIiwiQWNjb3VudCBTdGF0ZSI6InVuZGlmZmVyZW50aWF0ZWQifX1d
android.permission.CHANGE_COMPONENT_ENABLED_STATE	from checkPermission pid:2313,
internet.http.url	http://srgnhl.cs.illinois.edu/maldemo/endpoint_for_pii.php,
Read Phone Info	Requires READ_PHONE_STATE,
Read Phone Info	from checkPermission pid:2313,
Read Calendar	from checkPermission pid:2313,
Read Browser	from checkPermission pid:2313,
Read Browser	from checkPermission pid:2313,
Read SMS	from checkPermission pid:2313,
GPS Location	from checkPermission pid:2313,
android.permission.READ_CALL_LOG	from checkPermission pid:2313,

Figure 9: AndroMEDA detecting the Info Theft IncognitoWare embedded within a security app

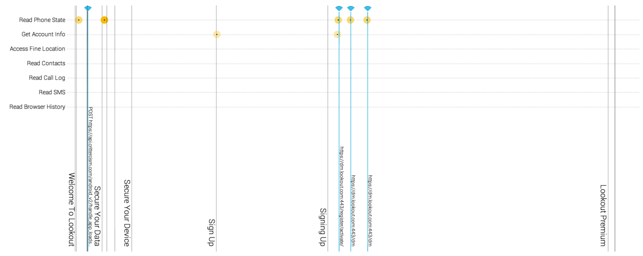


Figure 10: AndroMEDA logs of the normal version of the security app

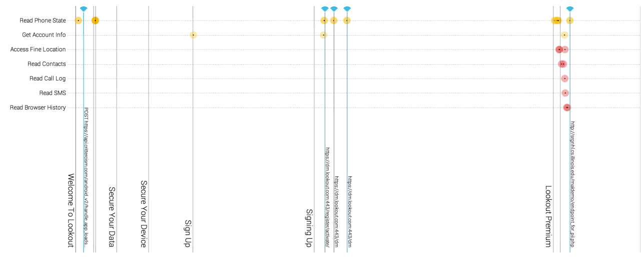
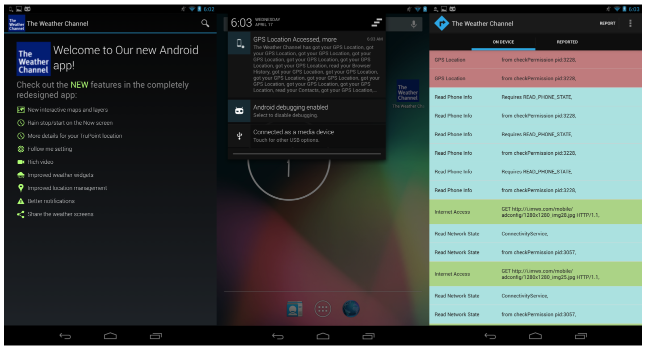


Figure 11: AndroMEDA logs of Info Theft IncognitoWare embedded within a security app



GPS Location	from checkPermission pid:3228,
GPS Location	from checkPermission pid:3228,
Read Phone Info	Requires READ_PHONE_STATE,
Read Phone Info	from checkPermission pid:3228,
Read Phone Info	Requires READ_PHONE_STATE,
Read Phone Info	from checkPermission pid:3228,
Read Phone Info	Requires READ_PHONE_STATE,
Read Phone Info	from checkPermission pid:3228,
Internet Access	GET http://i.imwx.com/mobile/adconfig/1280x1280_img28.jpg HTTP/1.1,
Read Network State	ConnectivityService,
Read Network State	from checkPermission pid:3057,
Internet Access	GET http://i.imwx.com/mobile/adconfig/1280x1280_img25.jpg HTTP/1.1,
Read Network State	ConnectivityService,
Read Network State	from checkPermission pid:3057,

Figure 12: AndroMEDA detecting the Spyware IncognitoWare embedded within a weather app

the requirements for spyware: access to location, Internet, wake lock, and starting at boot. We then introduce our malware, which is triggered an arbitrary amount of time after the initial startup — five minutes is long enough to evade Google Bouncer[27]. After that, the device wakes up every five 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 12 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 of this app, (Figure 13 — with context added), finds this to be a continuous occurrence, and decides that the app is malware. Compared to Figure 15, the app is clearly accessing location in a suspicious pattern — and the usage pattern is the key indicator. When compared to Figure 14, however, the main indicator is the address the information is going towards, an address not associated with the main app. The normal logs in Figure 14 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.

6.5 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, enabling them 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.

6.6 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. The ability 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.

7. 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 acting maliciously or not. In this paper, we introduced

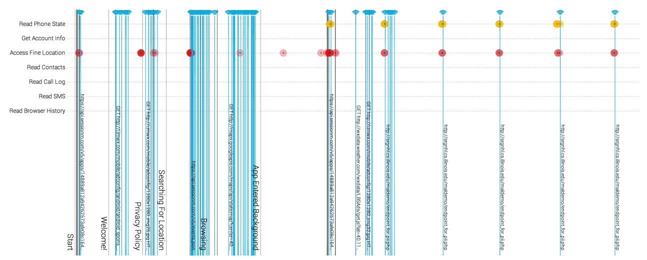


Figure 13: AndroMEDA logs of Spyware IncognitoWare embedded within a weather app

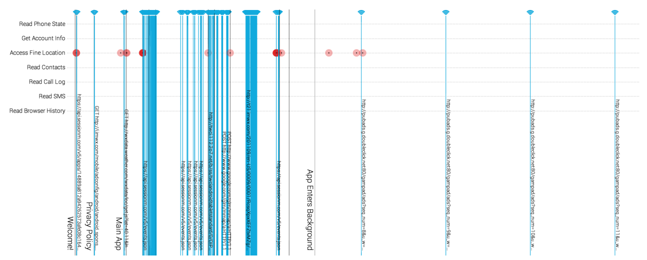


Figure 14: AndroMEDA logs of a normal version of a weather app, when the user has consented to location gathering

four key items: User-App Agreement, Android Census, AndroMEDA and IncognitoWare.

7.0.1 *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 generalized into the User-App Agreement — a framework for consenting to and trusting specific actions an app may perform. Whereas Android Permissions exceeded at defining general capabilities of an app, and these capabilities further shape the User-App Agreement, they fail to address the context in which the permissions are used and what they are used for.

7.0.2 *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 versus 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 is present, but not strong enough to differentiate between Info Theft Malware and many popular apps.

7.0.3 *AndroMEDA*

Building off the concept of the User-App Agreement, we introduce AndroMEDA. Key parts of the User-App Agreement were previously unavailable to the user until AndroMEDA. By giving the user more information on the context and use of permissions, they can evaluate whether or not they trust those actions and, ultimately, whether or not 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 increases user's awareness of app behavior and helps mitigate Info Theft Malware on Android.

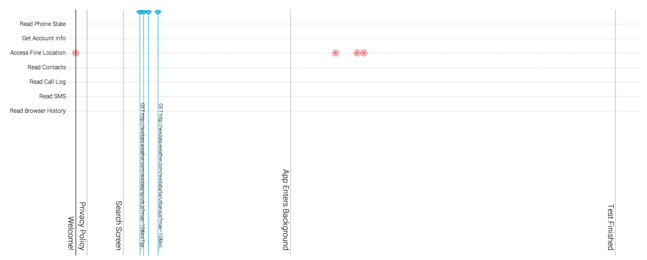


Figure 15: AndroMEDA logs of a normal version of a weather app, when the user has not consented to location gathering

7.0.4 *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.

7.1 Future Work

AndroMEDA is, ultimately, not a complete solution for 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 the AndroMEDA system to temporally block access to sensitive data, while unblocking 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 feedback to the user; 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 would instead be better able to dictate when to send different classes of alerts to the user — as the decision of what is malware is ultimately 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 could be greatly expanded upon. 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 demonstrate the effectiveness of AndroMEDA in spotting Info Theft Malware.

8. FUTURE WORK

THE FUTURE.

9. ACKNOWLEDGMENTS

References

[1] http://developer.apple.com/library/ios/#documentation/ContactData/Conceptual/AddressBookProgrammingGuideforiPhone/Chapters/UI_Controllers.html#//apple_ref/doc/uid/TP40007744-CH5-SW1.

[2] http://developer.android.com/about/dashboards/index.html.

[3] http://developer.android.com/about/versions/jelly-bean.html#42-security.

[4] http://source.android.com/.

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

[6] http://itunes.apple.com/.

[7] 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.

[8] 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.

[9] 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.

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

[11] http://www.cyanogenmod.org/.

[12] http://stats.cyanogenmod.org/.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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-07
2012.

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

[21] http://www.csc.ncsu.edu/faculty/jiang/GingerMaster/.

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

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

[24] https://play.google.com/store/apps/details?id=com.instagram.android.

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

[26] 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-dr
2011.

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

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

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

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

[31] Nq mobile 2013 security report. `http://www.nq.com/2012_NQ_Mobile_Security_Report.pdf`, 2013.

[32] 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.

[33] Industry leaders announce open platform for mobile devices. `http://www.openhandsetalliance.com/press_110507.html`, 2007.

[34] `http://www.openhandsetalliance.com/android_overview.html`.

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

[36] 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.

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

[38] 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.

[39] 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.

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

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

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

[43] 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.

[44] `http://m.trendmicro.com/mobileappreputation`.

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

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

[47] A. VARKOÄŇKOVÃĄ. DonâĂŹt think alternative markets save your money. `https://blog.avast.com/2012/05/14/dont-think-alternative-markets-save-your-money/`, 2012.

[48] `http://en.wikipedia.org/wiki/File:World_Wide_Smartphone_Sales_Share.png` accessed April 16, 2013.

[49] `http://en.wikipedia.org/wiki/File:World_Wide_Smartphone_Sales.png` accessed April 16, 2013.

[50] 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.

[51] 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.

[52] 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-r`
2011.

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

[54] 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.

[55] 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.

APPENDIX

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 1: Various properties of a Google Play Store app page