AppProfiler: A Flexible Method of Exposing Privacy-Related Behavior in Android Applications to End Users

Sanae Rosen
University of Michigan
Ann Arbor, MI
sanae@umich.edu

Zhiyun Qian
University of Michigan
Ann Arbor, MI
zhiyunq@umich.edu

Z. Morley Mao
University of Michigan
Ann Arbor, MI
zmao@umich.edu

ABSTRACT

Although Android's permission system is intended to allow users to make informed decisions about their privacy, it is often ineffective at conveying meaningful, useful information on how a user's privacy might be impacted by using an application. We present an alternate approach to providing users the knowledge needed to make informed decisions about the applications they install. First, we create a *knowledge base* of mappings between API calls and fine-grained privacy-related behaviors. We then use this knowledge base to produce, through static analysis, high-level *behavior profiles* of application behavior. We have analyzed almost 80,000 applications to date and have made the resulting behavior profiles available both through an Android application and online. Nearly 1500 users have used this application to date. Based on 2782 pieces of application-specific feedback, we analyze users' opinions about how applications affect their privacy and demonstrate that these profiles have had a substantial impact on their understanding of those applications. We also show the benefit of these profiles in understanding large-scale trends in how applications behave and the implications for user privacy.

Categories and Subject Descriptors

D.4.6 [**Security and Protection**]: Access controls

General Terms

Security

Keywords

android; smartphones; permissions; privacy

1. INTRODUCTION

The rise of mobile devices has lead to new concerns regarding application privacy and security. Not only are these devices now nearly as powerful and functional as personal computers, but they also carry detailed information about users' personal lives, such as their location, phone calls, and SMS messages. Much attention has been given to the threat of malware targeting these systems. Furthermore, these new capabilities mean that otherwise legitimate applications can become a significant privacy concern as well. Recent events, such as the outcry surrounding Carrier IQ [6] and concern over the Facebook application's behavior [14] suggest that the general public would appreciate being better informed about how the software on their phones impacts privacy. Furthermore, different users may have different expectations and needs with regards to privacy. Merely filtering out malicious applications is no longer sufficient; users also need to better understand the behavior of legitimate applications. Our goal is to produce a system that allows users to understand the privacy implications of applications they install by providing profiles of privacy-related application behavior.

The Android permission system is an important step towards addressing this problem, allowing users to make informed decisions by requiring that applications declare which capabilities they intend to use at install time. However, as we discuss in §2.1, this system has significant limitations. As the permission system is tightly integrated into Android, any substantial changes would require rewriting existing applications, meaning it lacks the flexibility needed to adapt to the rapidly changing world of application privacy. Furthermore, as it also serves as a capability enforcement mechanism, the descriptions it provides of applications are excessively broad, in order to meet the needs of developers. Because of these limitations, we argue permissions are the wrong abstraction to use in helping end users understand application behavior. We propose instead analyzing applications offline to create *behavior profiles*, separating the problem of understanding application behavior from that of capability enforcement.

To do this, it is necessary to automatically extract information about application behavior from applications. In traditional operating systems, work has been done on observing application behavior at a low level, e.g., by monitoring system calls [23]. This level of abstraction provides detailed and accurate information, but is hard to translate to something meaningful to users. The permission system provides very high-level information about application behavior, but is not specific enough to be useful, as we discuss in §2.1. In between the two, in API-driven systems such as mobile systems, applications access most sensitive information and functionality through API calls. In order to make use of these API calls, we start by building a *knowledge base* which consists of a series of *rules* translating API calls to application behaviors. We have identified 221 distinct application behaviors for which we have created rules. For example, we have identified four specific API calls that compare the user's distance to a given location. The mapping of these API calls to this behavior category is an example of a rule.

Given such a knowledge base, we create behavior profiles that provide more insightful information than the existing permission system. For example, consider an application requesting permission to access the GPS. The associated profile would indicate that the application is additionally concerned with the user's proximity to a location, and requests GPS updates at a rapid rate. An excerpt from a behavior profile, compared with the equivalent permission, is shown in Table 1, §4.2. To provide these profiles we use a second mapping, from matched rules to profile entries.

This two-step translation approach has significant advantages over the permission system as well. It gives a great deal of flexibility in how we present information. We can provide high-level profiles to non-technical users, but just as easily we can provide detailed technical information to security experts, using the same information extracted using the knowledge base.

We give users access to these profiles through an Android application and a web page. We do not address the problem of determining if an application's behavior is acceptable, leaving it to the user to make that judgment based upon their own requirements. For this paper, we have focused on Android, but in principle this approach would work for any system relying on API calls for access to important functionality.

Our work has the following novel contributions:

- A method of creating a *knowledge base* mapping API calls to application descriptions, and then using this knowledge base to create *profiles* of application behavior.
- An efficient application of this method to the large-scale, automated analysis of applications in Google Play. We currently analyze an average of 500 applications per day on a single server, and our approach is completely parallelizable. So far we have analyzed almost 80,000 applications.
- A large-scale survey of the privacy- and security-relevant behavior of a significant cross-section of Google Play, as well as user perceptions of these behaviors. Some of our findings include determining users are concerned about behavior which is most prevalent in ad libraries, and determining that many applications are less intrusive than they appear from the permissions they request.
- We identify a number of ways in which permissions do not provide information that users care about (e.g., user-triggered SMS messages vs. those occurring in the background) and offer suggestions to improve the permission system.

The paper is organized as follows. §2 summarizes related work in this area. We provide an overview of our approach and threat model in §3. In §4 we discuss how to create and use the knowledge base, how to make this analysis scalable and automatic, and how we make the results available to the public. In §5, we examine how well our system performs against a variety of application types, examine a number of prominent applications in depth, look at large-scale trends in application behavior, and examine how users of our AppProfiles application make use of our profiles.

2. BACKGROUND AND RELATED WORK

We first give some background on the Android permission system followed by a summary of related work.

2.1 Android Permission System and its Limitations

In the permission system, applications declare the capabilities, or "permissions", they intend to use at install time. Permissions cover broad classes of functionality, like "Internet" or "Read Phone State", and they must have been declared for an application to access that functionality. There are several issues with this approach.

1. Some permissions are so prevalent that users are likely to ignore them, such as the Internet permission [4, 17].
2. Permissions generally cover broad, vague categories of functionality and give insufficiently detailed information for users to make meaningful decisions [15]. For example, Read Phone State covers everything from reading the phone number to reading the OS software version [16].
3. Applications often request permissions they don't use, so permissions don't necessarily correspond to application behavior [16, 4].
4. Occasionally behavior which should be protected with a permission is not, allowing applications to bypass the permission system [26, 21].
5. If a problem is found with the permission system, it is hard to fix due to tight integration with the Android system, making problems 1, 2, and 4 hard to address. This results in a trade-off between OS stability and fine-grained permissions [3].

These issues suggest that the permission system's ability to provide useful information to end users is limited.

2.2 Related Work

Previous work has approached the problem of understanding application behavior in different ways. One much-studied area is the permission system. It has significant limitations, and many papers have explored or attempted to address these limitations. Recent work by Grace et al. [22] detects mechanisms by which permissions granted to one application can be leaked to another, either inadvertently or deliberately through collusion. The Apex system [30] proposed by Nauman et al. illustrates that the permission system may be too broad to be useful, and implements a more sophisticated permission system that allows users to limit the scope of key permissions. Roesner et al. [32] present a permission system where users directly grant permissions to applications at runtime in a non-intrusive manner by integrating permission granting with existing UI elements. Extensive misuse of the permission system by developers has been identified by Barrera et al. [4]. Kirin [12] detects potentially dangerous applications at install-time based on combinations of permissions and intent strings. TISSA [36] gives users more control over how applications with permissions access their data. However, any change to the permission system would require fundamental changes to Android, which may limit how likely these solutions are to be implemented.

Other work seeks to understand applications at the system level. The use of low-level features like system calls to detect anomalous behavior is a technique that has been used in traditional operating systems [23], but it has also been successful for Android. Crowdroid [5] detects malicious applications masquerading as other applications using Linux-level system calls to detect anomalous behavior among applications with the same name and version number. Information gleaned at this level is more accurate and fine-grained than at the permission level, but it is likely not of direct use to the average user in understanding application behavior.

The limitations of these two approaches — permissions and system-level information — suggest that it is worth looking elsewhere for a solution. The Android Framework API has the benefit of providing extensive and accurate information, like the system layer. We show that it can also be used to accurately and flexibly emulate the kind of high-level profiling the permission system attempts to do. We leverage results from several papers to accomplish this. Work by Felt et al. [16], and PScout, by Au et al. [3] maps API calls to permissions and allows developers to determine what permissions they should use. We have made extensive use of their data set as a starting point to understanding the Android API. Unlike them, however, we address the problem that permissions may not provide sufficient information to users. Our work also builds upon a paper by Enck et al. [11], which demonstrates the feasibility of using existing static analysis tools for Java to detect malicious behavior in Android applications. Similarly, Lu et al. [29] have constructed a system to statically detect certain Android-specific security vulnerabilities. Unlike these papers, we take the approach of creating a comprehensive picture of privacy-relevant application behavior rather than identifying specific instances of malicious or vulnerable behavior.

Several other studies have developed complementary mechanisms to improve user and developer control of applications. For example, AppFence [24] introduces innocuous *shadow data* to replace sensitive data and introduces new privacy controls for existing applications. Quire [9] provides a framework to allow applications to determine the call chain of requests made, allowing applications to protect themselves from other malicious applications. A recent study by Szydlowski [34] explores the feasibility of detecting malicious behavior dynamically in iOS applications. Finally, TaintDroid, by Enck et al., [10] modifies the operating system to track the flow of sensitive data to detect when this data is exfiltrated. All of these have goals orthogonal to ours, however.

3. OVERVIEW

In AppProfiles, we create descriptions of security- and privacy-relevant behavior which enable users to make informed decisions about what applications to allow on their phones. As the Android API imposes a structure on how applications access sensitive information or functionality, we leverage it to detect behaviors of interest using static analysis (with a few limitations, as described in §5.1). This approach is complementary to the permission system and does not attempt to replace existing malware detection tools such as antivirus software. We do not seek to constrain application behavior, the way the permission system does, or to detect malicious applications that attempt to subvert the constraints imposed by the Android API.

In order to do this, we start by creating a *knowledge base* of mappings between API calls and application behavior types. Such a mapping necessarily involves some degree of manual effort, but there are several tools and techniques we use to minimize the effort involved. We start with a list of key security- and privacy-relevant API calls, which we map to behavior descriptions. Existing research [16, 3] allows this mapping to be constructed easily and automatically. Next, we identify API calls which benefit from having more specific entries in our knowledge base associated with them. For example, it may be valuable to detect if certain arguments are passed to a given API call. We describe in § 4.1 how to refine our rules more systematically. Borrowing from the terminology used by our static analysis tool, we call each entry in the knowledge base a *rule*.

Next, this knowledge base is used to create *application behavior profiles*. We use an existing static analysis tool to detect the code patterns in our knowledge base. We translate this data into profiles using a second mapping between rule matchings and higher-level descriptions, which are intended to be accessible to end users. This mapping to high-level descriptions also allows behavior to be inferred from combinations of rules triggered. For example, consider an application where a rule involving taking a photograph has been triggered, but a rule for showing the user a preview of the image has not been triggered. The corresponding behavior profile entry would indicate to the user that the application is capable of taking photographs without displaying anything on the screen.

There are two major concerns in producing our knowledge base: accuracy, and completeness. As there is no ground truth to compare against, these are hard to evaluate. However, as we use permissions as a starting point and then add a significant amount of additional data, it is necessarily more complete than the permission system. Furthermore, the feedback we have collected indicates that users have found the new information gathered to be substantial and useful (see §5.4). We have also added rules based on security threats from known malware, as anti-viruses do (for example, sending SMS messages without user input), but unlike them we can cover behavior that is not explicitly malicious and which requires human judgment to evaluate (for example, collecting different types of location data.) As new privacy threats become apparent we can easily add new rules. We evaluate the overall accuracy of the entire system in §5.1.

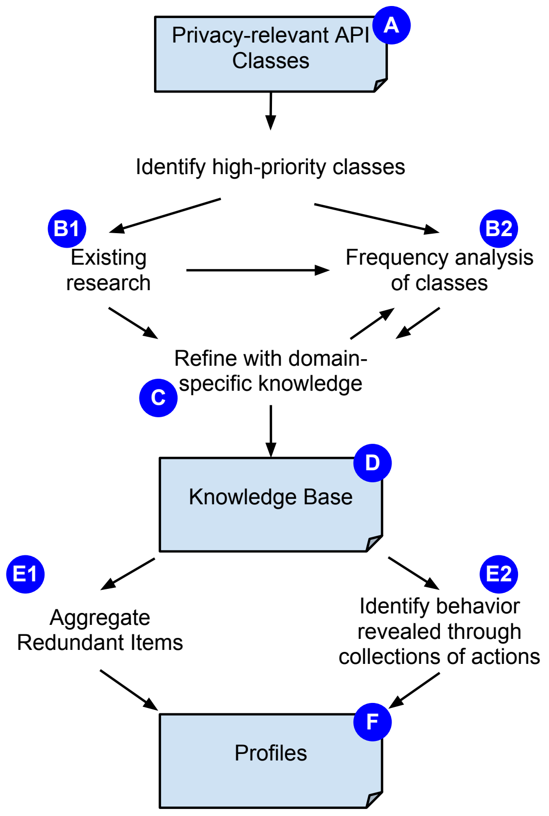
An important feature of this approach is that we have separated the collection of data (using our knowledge base) from the interpretation of this data (as profiles). This allows for a wide range of different types of profile information to be created without modifying the data collection method. We only provide information to ordinary users which is designed to be comprehensible and of relevance in trying to determine whether to use an application. However, advanced users might want to have access to more technical details. These advanced profiles also allow us, and potentially other researchers, to perform further analysis. We give some examples of the types of analysis enabled by these profiles in §5.2 and §5.3.

The final step is to use these profiles to evaluate whether this application is appropriate to run on a given device. This will vary with the needs of the user, so we allow users to make that decision based on their needs. As we have been collecting feedback from users regarding their views on application behavior, future work could include making an application privacy rating available to users based on the profiles as well as user feedback.

We emphasize that our focus is on allowing users to make informed decisions about legitimate applications, not on detecting malware. We do not attempt to deal with native code or applications that take extraordinary steps to prevent analysis — we are focused on detecting how the Android API is used and anything that is external to this API, or that attempts to subvert its protections, is outside the scope of this work. We also do not attempt to determine what is an unacceptable privacy violation. Much of the behavior we discover may, in context, be harmless. For example, an app that covertly tracks a phone in case it is stolen is indistinguishable from spyware. Others may differ from user to user. For example, not all users may be concerned with applications that track their location. The goal of AppProfiler is to give users the ability to make informed decisions about the applications they install, something which we believe cannot currently adequately be done.

4. DESIGN AND IMPLEMENTATION

There are four significant aspects to the design of the system. First, we describe how we create our knowledge base. Second, we explain how this knowledge base is used to generate behavior profiles. Third, we discuss how to develop a scalable system to produce such profiles for the available applications in Google Play. Finally, we discuss how we make these profiles readily available to end users.



A
Privacy-relevant API Classes
Identify high-priority classes
B1
Existing research
B2
Frequency analysis of classes
Refine with domain-specific knowledge
C
D
Knowledge Base
E1
Aggregate Redundant Items
E2
Identify behavior revealed through collections of actions
F
Profiles

Figure 1: Conceptual overview of profile creation process.

4.1 Creating a Knowledge Base

We wish to identify a set of privacy-relevant API calls and map them to appropriate behavior types which summarize the privacy implications of those calls. For example, if the API call "android.telephony.SmsManager.sendTextMessage" is detected and its first argument is a constant, we wish to map this to the behavior, “sends text messages to a fixed phone number". As the range of these possible mappings is very large, a method is needed to systematically create rules and determine which API calls to detect. We use the Android permission system to do some of the work, taking advantage of research [16, 3] on automatically mapping permissions to function calls. While this is sufficient for our system to function, substantially more detailed information can be provided by a deeper examination of the Android API.

Our approach to enhancing the knowledge base in this manner is summarized in Figure 1. We start by creating a list of rules that match very broad types of behavior, mostly at the level of classes (step A in the figure), based on existing API-to-permission mappings, as well as through manual effort. As the API classes are well-documented and clearly-named, manually identifying which additional classes are of interest required little effort. We then increase the quantity and specificity of the information we provide. This also requires some manual effort, but this can be significantly reduced by first determining which parts of the API would benefit from more detailed analysis, then covering the remaining, less interesting, parts of the API with more general rules.

To do so, we first refer again to previous research to identify key areas of the API (step B1). For example, previous work [15, 4] suggests that the Location, Internet and Phone State permissions are among the most commonly used, as well as highly privacy-sensitive. Therefore, adding more detail in these areas would allow users to better understand a wide range of applications.

We also ran additional tests to identify and confirm which classes are significant (step B2). We ran a preliminary version of our profiler on a selection of applications, including popular and randomly selected applications from several markets, as well as a selection of malware [7]. By doing so we confirmed the results of existing research and identified other areas in which more detail would be helpful, focusing on more common or dangerous behavior.

Once we narrow down a number of classes of interest in this manner, we manually examine the methods of each class and determine which are of potential significance to users (Step C). For example, detecting the API call that reads the phone number of a device, or reads device-specific IDs, is likely to be of interest to users. Detecting the API call that specifies a format for the name of the network operator, prior to retrieving that name, likely is not. For large, complex parts of the API we iterate over steps B2 and C to further refine our knowledge base.

Most of our rules deal with short, specific API patterns, such as function calls and their arguments, or combinations of function calls, as it makes it easier to build up complex profiles from this information later.

To make this process more concrete, here is an example of how two sets of rules get developed.

1. Using an existing mapping of permissions to API calls (step A), we identify which classes use the "Location" permission. We spend more effort here because previous research suggests location data is important to users (step B1). We create a preliminary set of *rules* that match any use of the Location classes.
2. After running these rules on a preliminary set of applications, we determine that the LocationManager and Address classes, among others, are widely used (Step B2).
3. We examine the methods of these classes. We identify those that look interesting: these include a series of methods in "LocationManager" concerned with how often location updates are given. In "Address", there are a series of methods concerned with various types of address data, ranging from the user’s country to their street address. We create a new set of rules that match with each of these API calls (Step C).
4. We use the results of the rule matchings to decide where more detail is needed.
    - We determine that many of the specific functions from the Address class are rarely used, so for our final set of rules we combine similar functions. For example, we place all those related to human-readable, fine-grained addresses together (e.g. getAddressLine, getThoroughfare), and we place all those pertaining to the user’s country together (e.g. getCountryCode, getCountryName). We are now done with creating the set of rules for “Address." (step D).
    - Conversely, many of the specific API calls in LocationManager are very common. Therefore, it is worthwhile to expend more effort in creating more specific rules to describe these, and so we iterate again over step B and C. For example, we notice “requestLocationUpdates" is very common. It takes arguments determining how often these updates happen, which may be of interest to users, as the frequency of updates impacts their battery life. We create rules for different update frequencies and again run these rules on our set of applications.
5. Finally, we notice a lot of applications retrieve location data at the maximum possible rate. We manually examine a few,

and determine that what is actually happening is they request repeated updates, but stop the request as soon as they have adequate data. Older versions of the API did not allow a single location update to be requested. We treat this pattern separately so as to not claim that applications which do this are wasting battery. We now can add these rules to our knowledge base (step D).

As a result of this process, we have now identified twelve rules for our knowledge base. One, associated with LocationManager, identifies if the method “requestLocationUpdates" has been used at all. Seven more narrow down the frequency of these updates. Four are associated with “Address": one checks if the application requests information about the user’s country, another about the user’s state, another about the user’s postal code, and the last about the user’s street address. Several more rules are associated with these classes: the full knowledge base can be seen at `http://appprofiles.eecs.umich.edu/tech.html`. We give an example of the knowledge base entry for checking the user’s state below:

```
Category:
Location - Type
Subcategory:  Regional data - State

FunctionCall call:
call.function.enclosingClass.name startsWith
"android.location.Address"
and call.function.name == "getAdminArea"
FunctionCall call:
call.function.enclosingClass.name startsWith
"android.location.Address"
and call.function.name == "getSubAdminArea"
```

4.2 Developing Profiles

This knowledge base is then applied to better understanding application behavior. We start by extracting the API calls in our knowledge base from the application source code. In order to do this, we use a tool called the Fortify Static Code Analyzer [18], which is able to use the rules in our knowledge base to identify code in each application that matches our rules.

Once we have a complete list of which rules in our knowledge base have been triggered for a given application, we process this data in order to produce an easily understandable final product. We combine redundant rules (E1) and detect behavior inferred from the total set of rules (E2). Some forms of analysis are best done by inferring results from several rules. For example, we wish to know if a SMS message was sent in the context of a background service. We have one rule for detecting SMS messages being sent, and another for detecting classes that run as background services. By looking at where the former rule was triggered, if it occurs in a class that runs as a background service, we know it happened in the background. In fact, we can then determine if any rule is triggered in the background in the same way.

To make this approach more concrete, here is an example of how this process occurs.

1. We run the rules against the application, which reveals which rules are triggered and in which classes they occur. Rules are triggered which indicate:

    - The application asks for the fine-grained or coarse-grained location, whichever is most recent.

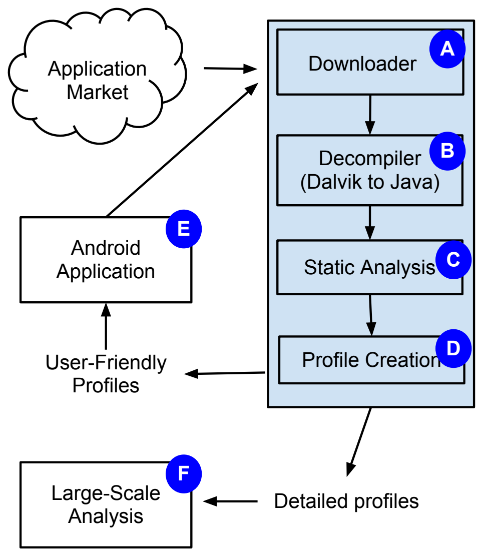


Figure 2: Overview of the application analysis system

    - Location updates are requested every five minutes in one location; every ten minutes in another.
    - The application requests that the distance to a fixed location be calculated.
    - The phone’s IMEI is requested.
    - We also identify, where possible, which classes are run in the foreground and which run as background services. Looking at which methods are called by classes in other files sometimes does not work reliably.

2. We organize the results of these rule matches, incorporating data on whether classes run into the foreground or background, as well as data on which classes belong to which ad library (Step E2). Of the two rules indicating how often the application requests updates, we only report the worst-behaving one (Step E1).

    - The application selects the best of the fine-grained or coarse-grained location; this happens in the background.
    - The application requests updates every 5 minutes (this being the worst-case).
    - The application cares about proximity to a location; this occurs in the foreground.
    - A unique, personally-identifiable ID is requested by Google’s ad library. We cannot determine if this happens in the foreground or background.

Additionally, we adjust our profiles based on our audience. A technical user might be interested in knowing which phone-specific IDs are being tracked, for example, whereas a more casual user might only care that they are being tracked. As our profiles are created by a script that maps combinations of triggered rules to higher-level descriptions, it is straightforward to create different types of profiles by changing this mapping. Essentially, we separated data collection from processing, analyzing, and displaying the data to gain flexibility in the amount and nature of data that we present to various audiences. This also made it easy to add functionality as needed. For example, we added the ability to determine if behavior originates from third-party libraries without requiring substantial changes to our system.

Table 1: Comparison of feedback types with the associated permission in a sample app. We use the following notation: <text> indicates the context in the application in which the rule is triggered, such as in an activity (i.e., the foreground). [text] indicates an associated major ad library.

Permissions	Our User Summaries	Our Technical Summaries
• fine (GPS) location - Access fine location sources such as the Global Positioning System on the phone, where available. Malicious apps may use this to determine where you are, and may consume additional battery power.	• Gathers fairly precise location data (e.g. GPS) • Reads your latitude or longitude • Uses more of your phone's resources than recommended by Google to retrieve your location data. • Concerned with your proximity to a given location (for example, may be alerted if you are near a particular store.)	• Can use GPS OR network <context unknown> • Latitude/Longitude <broadcast> [googleads] • Updates every 1s or less <activity> [jumptap.adtag] • Asks for periodic updates <activity> • Proximity to location <on click> <activity>

We currently produce two forms of output (see Table 1). One is more technical and detailed, and primarily for the benefit of researchers and security professionals. We used this to perform analysis on trends in application behavior, such as comparing the behavior of third-party code with application-specific code. We plan to make it available in the near future. The second is simpler, and is aimed at security-conscious but non-technical end users. We have made these profiles available through an application in Google Play which to date has seen almost 1500 downloads.

4.3 Large-Scale Application Analysis

Next, we demonstrate that these profiles can be used, in the long term, to analyze all the available (i.e., free) applications in Google Play.

We start with an existing list of all applications in the market [27], containing 276,016 apps, around 45% of which appear to be no longer be accessible. Building off of the Unofficial Android Market API [1], a script downloads these applications. (This is step A in Figure 2).

For the next stage (step B in Figure 2) we use ded [11] to decompile each application. Once we produce the source code, we analyze it using Fortify (step C) and the rules we derived as described above. We pass the resulting data to another set of scripts, which uses the set of rules triggered to create higher-level descriptions of application behavior (Step D).

We processed over 33,000 applications in 67 days on one server, a time period which included a number of interruptions as we tweaked our system. We analyzed on average about 500 applications per day. An additional 27,000 applications were queried but unreachable. We investigated a number of cases to confirm that there was no application with that package name currently active in the market, and that likely the application had been removed. Using a few servers of various speeds, we have already processed 65% of our app list. Furthermore, the system could be further optimized. For example, a system using a custom static analysis tool operating on disassembled Dalvik bytecode would be able to eliminate the time spent decompiling applications, which we found accounted for over 90% of the time spent.

There are a number of technical limitations due to artifacts of how we implemented the system. The decompilation process is not perfect [11]. We examine some randomly selected applications in §5.1.1 to better understand the impact of these errors. Alternately, the application could be disassembled using a tool like baksmali [33], although in that case a custom analyzer would also need to be built.

We make results available through an android application, *AppProfiles* (Step E in Figure 2). It allows users to select an application to look up from a list of applications installed. We give users a less technical version of the profiles (see Figure 1 for a brief example.) We also allow users to submit feedback on application behavior. They can toggle actions they don't like and indicate whether the application behaves as expected overall. Since we download applications from Google Play and not the user's phone we are limited to free applications. We additionally made the profiles viewable on our website using a similar interface, at `http://appprofiles.eecs.umich.edu`.

5. RESULTS AND ANALYSIS

We present several types of analysis below. First, we measure the accuracy of our system and identify any limitations that may impact its accuracy. Second, we examine three applications in depth to better understand these limitations, as well as demonstrate the sort of information our profiles make available. Third, we look at how our profiles can be used to quickly and easily gain an understanding of the market as a whole. We compare trends in library-specific and application-specific code and look at how popular applications differ from other applications. Finally, we examine feedback that users have given us about our profiles and what they indicate about the applications they run.

5.1 Testing Profile Accuracy

In order to evaluate the accuracy of our profiles, we first examine a set of popular applications and of one of randomly selected applications, which we examine manually to determine what our profiles should detect. We also examine a set of malware that has well-studied behavior as a worst-case analysis of the accuracy of our system.

5.1.1 Confirmation through dynamic analysis

In order to measure how often the behaviors we detect occur in practice, we created a dynamic analysis testing framework based on TaintDroid [10], with the added ability to detect behavior corresponding to that found in our profiles. This testing framework is somewhat limited, as certain types of functionality cannot be detected by this dynamic testing system. For example, the technical version of our profiles distinguishes between applications using

HTTP libraries and those using TCP libraries, but the former results in the latter being called at runtime. Nevertheless, it helps illustrate the limitations of our system.

We analyzed a selection of two types of applications: popular applications and randomly chosen ones. For the latter, we excluded two that would not run at all, and two where language barriers prevented us from understanding the application's behavior and thus prevented us from thoroughly exercising the application. We compared against our more technical profiles, but when accounting for the accuracy of these profiles, we combined similar rules regarding Internet use, as these are very common, often appear together and are usually correctly detected — they would tend to artificially inflate the accuracy of our profiles.

We selected 10 popular applications from the front page of Google Play. They cover a range of different application types and publishers. Overall we found that applications that make extensive use of third-party libraries may have higher rates of false positives. For example, one application uses a debugging library whose behavior is not triggered at runtime but whose source code is included in the APK. This is an inherent problem with static analysis — however, using static analysis allowed us to cover far more applications. Conversely, errors in decompilation can result in behavior not being detected in our rules. Given the limitations of our testing framework, any values on accuracy should be taken with a grain of salt. However, an average of 10% of behavior triggered dynamically did not appear in our profiles; we had a 23% false positive rate.

We selected 15 applications at random, discarding four for the reasons described above. We had a 16% false positive rate and 15% false negative rate; most false negatives were caused by a single application which failed to decompile correctly. The lower false positive rate may be due to these applications being much simpler and easier to exercise thoroughly. For both application sets, there was no clear pattern in the behaviors we missed, as these appear to be due to decompilation errors or the use of native code.

5.1.2 *Testing Malware With Known Behavior*

We would like to reiterate that the goal of this project is not to detect malicious software, and in particular not to detect malicious software that takes extraordinary steps to hide its behavior from researchers. Nevertheless, examining the behavior of known malware is useful for several reasons. First, many malicious applications have been well-studied and analyzed by a third party, which means that a ground truth for their expected behavior exists. The same is not true of most legitimate applications. Furthermore, as many of them take extensive steps to obfuscate their behavior, they allow us to do a worst-case evaluation and determine the limitations of our system. We downloaded the entire Contagio mobile malware sample on January 7, 2011, and randomly selected a number of applications to analyze in detail (see Table 2). We could find a description of 28 of these randomly selected applications from a security researcher or organization [28, 25, 19, 35]. Overall, our profiles detected an average of 59% of the expected behavior.

SMS-related behavior was one of the most common and serious behavior types we detected, and we were able to detect this in applications that rely on root exploits or native code. We also frequently detected these applications collecting location data and unique phone IDs, and using the Internet, although legitimate applications do this as well. Frequently, suspicious behavior occurs in the background (i.e., in Services or broadcast receivers). Our highest rate of failure was in detecting applications that download additional binaries, likely because malware does not generally use the provided API for doing so.

Table 2: Accuracy of our profiles for 28 malware samples. The last row applies to apps that exhibit suspicious behavior in a background application component. The false positive rate is likely an overestimate as security researchers may have felt some behavior is not worth listing.

Behavior Categories	Detected correctly	False Negatives	False Positives
Suspicious SMS	9	1	1
Normal SMS	0	0	0
Phone calls	1	1	2
Interacts with other apps	1	1	4
Downloads/installs apps	1	11	0
Collects list of apps	1	1	1
Runs Linux commands/root	4	2	1
Collects location data	8	0	4
Collects phone IDs	16	1	2
Adds bookmarks	0	1	0
Uses the Internet	16	0	2
Behavior occurs in background	17	N/A	N/A

We may also have detected some previously unknown behavior in some of the malware samples. In a few cases, the detected behavior was entirely different from the behavior we expected based on the behavior profiles written by security experts. Not only did we fail to detect the expected malicious behavior, but we also detected previously unknown malicious behavior. We manually verified, by examining the application source code, that our profiles in these cases matched the malware's actual behavior. A likely explanation is that these applications were mislabeled.

Overall, while the accuracy of our profiles is somewhat limited when it comes to malware, and we specifically did not aim to be able to detect malware with this system, we were nevertheless successful more often than not, and we have furthermore been able to detect previously overlooked behavior in malicious applications.

5.2 Case Studies

We have chosen three popular applications to analyze in depth in order to demonstrate the behavior these profiles can reveal. Two were found by our users to have a great deal of concerning behavior, and one was overwhelmingly rated as harmless.

5.2.1 *Facebook [13]*

As many of our users were concerned about the behavior of the Facebook application, we selected it for detailed examination. In our profile, we predict it can exhibit intrusive location behavior. It asks for the user's latitude and longitude and proximity to other locations. It also accesses video functionality as well as information about the carrier and the phone number. Based on our survey results, users seemed especially concerned about its ability to access their phone number and location, including the fact that it appears to query their location very frequently. On the other hand, they were not concerned about its use of the Internet.

Our technical profile suggests that the location related actions and those related to the user's phone number happen in an "Activity", i.e., a component the users interact with directly, whereas much of the Internet-related functionality happens in a background process with which users do not interact. Our profiles did not detect certain behavior for which permissions are requested; in particular, we detected nothing related to SMS messages. We used our dynamic testing tool to determine that our profiles were correct, at the time of analysis, and the application is over-requesting permissions.

Although its permissions and our user-friendly profiles paint the application as being overly aggressive in its use of privacy-sensitive functionality, it is less intrusive than it might seem, as much of its controversial behavior is only triggered by the explicit actions of the user. Nevertheless, our profiles are more accurate than the permissions in determining its behavior. This suggests determining the context of privacy-affecting behavior may be worth including in our more user-friendly profiles — we may need to expand this part of our analysis to be able to determine this context in every case.

5.2.2 *Angry Birds [2]*

This is another application which is both popular among Android users in general and distrusted among users of our application. As with Facebook, users seem particularly concerned about its location-related behavior; they also object to the use of cookies and personally identifiable phone identifiers. Unlike Facebook, this application appears to make use of native code, which we cannot analyze. Nevertheless, we were able to detect much behavior of concern.

Our profiles predict that the application itself is not very intrusive. However, like many free applications it contains ad libraries (we detected code from five) and these ads are responsible for much of the privacy-intrusive behavior. Dynamic testing confirmed this trend. It also determined that a great deal of of information is written to the debug log, including personal IDs and information about the carrier. The only discrepancy between our profiles and our testing is that the latter did not detect any use of the telephony manager or the sensor manager, behaviors which were in our profiles.

This example indicates two important features of our analysis. First of all, false positives will always be an issue in static analysis, as code may be present but never triggered. Secondly, our results confirm that a significant amount of concerning behavior occurs in ad libraries. This is a strong reminder that ad-supported applications might have a hidden cost in terms of privacy.

5.2.3 *Reddit is Fun [31]*

We chose this application as it was the most frequently ranked as having acceptable behavior by users and we wish to determine how a positively ranked application might differ from a negatively ranked one. The permissions, and the author's writeup, indicate that it should access the Internet, store data on the SD card, access the network connectivity state and start automatically on boot. Our profiles suggest that it can additionally update the user's browser history, process phone numbers in some manner, and use cookies. It accesses the Internet using both Webviews (a library to present information to users) and direct HTTP connections. All the behavior in our profiles was confirmed to exist through our dynamic verification framework.

The biggest difference between this application and the above ones is that this one does not collect location data, which as we discuss later seems to generally be of great concern to users. Furthermore, the developers explain in detail in the market writeup what the application does, which may make users feel more comfortable with its behavior.

Table 3: A comparison of types of behavior seen in the top 9 most common third-party libraries.

Behavior type	Google Ads	Google Analytics	Facebook	Admob	Millenialmedia (ads)	Flurry (ad/analytics)	Twitter4j	Phonegap (app platform)	Mobclix (ad/analytics)
Internet	*	*	*	*	*	*	*	*	*
Internet - webview	*		*	*	*	*		*	*
Location - passive	*	*		*					*
Location - active								*	*
Audio manager	*			*				*	
Hardware sensors					*			*	
Cookies		*							*
Camera								*	
Unique ids								*	
Phone number								*	
Bookmarks									*
Detect other tasks									*

5.3 Large-scale Analysis of App Behavior

5.3.1 *Third-Party Library Use*

Given that many applications make extensive use of third-party libraries, and prior research suggests that ad libraries are often quite intrusive in terms of privacy [20], we examine these more closely. First, we used a simple heuristic to identify these libraries. We counted the occurrence of every class name and its associated package name, keeping track of duplicates. As a common code obfuscation tool seems to also produce identical class names and package names across applications (e.g., a.b.java) we excluded class names fitting such a pattern. We then created a list of frequently repeated classes, using 100 unique instances across distinct applications as a cutoff. While this might exclude some minor libraries, it should cover any of significance. We then reran our behavior aggregation script, only this time dividing behavior into that unique to the application and that originating from a third-party library.

The behavior of the top 9 are summarized in Table 3. The overall trends — the widespread use of location data and personally identifiable information — is consistent with existing research [20]. There is a lot of variation in the behavior of these libraries, but in general those that merely integrate with an existing service (like Facebook and Twitter) are fairly non-intrusive, whereas ad libraries tend to be more intrusive. However, many of the top ad libraries are less intrusive than those seen in our case studies. This suggests it may be possible for ad libraries to meet user privacy expectations while still remaining commercially viable. Phonegap is an unusual case. Since it is a platform for assisting in creating applications it exhibits a wide range of behavior.

We also compared the behavior common in our total set of third-party libraries with that which is common in regular applications. We have show the most common behaviors from both cases in Figure 3. Sometimes, their behavior is quite similar — both use webviews a great deal (the standard API for rendering a webpage from a URL) and Internet-related behavior in general is common for both. In other cases, they behave quite differently. Third-party

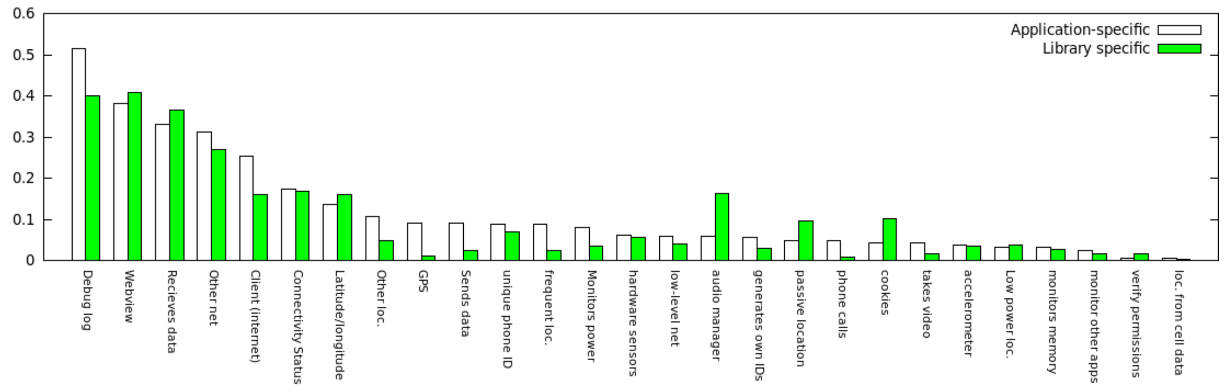


Figure 3: Comparison of the most common behavior in application-specific code versus that in third-party library code.

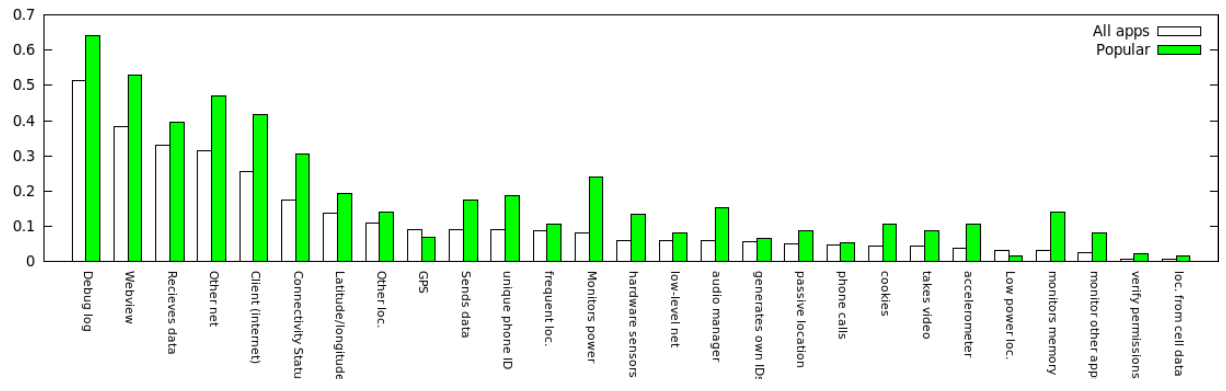


Figure 4: Comparison of the most common behavior for all apps versus that in 170 popular apps (application-specific code only).

libraries make greater use of cookies, for example. It is common for either to access a user's location, but there are some fundamental differences in how they do it. Third-party libraries are more likely to ask for a cached location, and applications are more likely to specify that they need to use the GPS exclusively. Overall, our results are consistent with existing research [20].

5.3.2 Popular Applications

It is also interesting to observe how the most popular applications compare with all other applications. A comparison can be seen in Figure 4, which shows application-specific code only— library-specific code exhibits the same trends. Of our total set of apps, at least one rule has been triggered in application-specific code for 66% of applications, and in third-party library code for 55% of applications (note that not all apps use third-party libraries). Of the popular applications, 74% have relevant behavior in application-specific code, and 79% in third-party code. This trend is reflected in Figure 4, where it can be seen that more popular applications exhibit all but two types of behavior more commonly than average. This difference is more pronounced in behaviors related to monitoring the system status; popular applications are more likely to monitor memory usage, power state, connectivity state, and the status of other applications. These differences suggest that research focusing only on the most popular applications may not reflect behavior in the market as a whole.

One other interesting fact is that the most common behavior detected in all cases is to print to the debug log, something which Google recommends that developers turn off in published applications [8]. This could leak important information, such as a user's private information, as we found in our case studies.

5.4 User Feedback from AppProfiles

The release of our application has demonstrated that there is interest in better understanding what smartphone applications do. Within the first week, we had 833 downloads, and a total of 1482 at the time of writing. Feedback has been positive, with an average rating of 4.1. Users have indicated that they feel such an app is much-needed.

We gave users the option of anonymously submitting feedback on the applications they use (see Figure 5). These results are not representative of the entire population of smart phone users, as the findings are likely to be skewed towards those who are interested in security and privacy issues. However, as the use of this application is entirely voluntary and only useful to the privacy-conscious, this is arguably the demographic whose responses are most relevant.

We surveyed the feedback submitted roughly a week after uploading our application. At the time, we had 1839 distinct items of anonymous feedback covering a total of 456 unique applications. 63% of those applications were ranked as acceptable by at least one user, 28% as exhibiting surprising but acceptable behavior, and 43% as exhibiting unacceptable behavior.

The most commonly rated applications are summarized in Table 4. Ad-driven popular games and social media applications seem to be particularly controversial, likely for the reasons discussed in §5.2. It was also fairly uncommon for an application to exhibit unexpected behavior without users also considering that behavior objectionable. The least controversial applications tend to be work-oriented or provide some basic utility, such as Dropbox, Google Docs or Flash Player. The "reddit is fun" application is an exception, perhaps because it is not ad-oriented.

Table 4: Top most common applications in each category by number of ratings; percent of feedback items in that category also shown for each app.

Acceptable		Surprising		Not Acceptable	
com.andrewshu.android.reddit	56 (90%)	com.facebook.katana	13 (14%)	com.facebook.katana	53 (58%)
com.dropbox.android	40 (83%)	com.pandora.android	6 (26%)	com.zynga.words	31 (86%)
com.alensw.PicFolder	27 (90%)	com.amazon.kindle	6 (22%)	com.rovio.angrybirds	29 (91%)
com.bigtincan.android.adfree	22 (88%)	com.devuni.flashlight	5 (13%)	org.zwanoo.android.speedtest	19 (68%)
com.adobe.flashplayer	21 (95%)	org.zwanoo.android.speedtest	5 (18%)	com.imdb.mobile	13 (76%)
com.google.android.apps.docs	19 (100%)	com.evernote	4 (23%)	com.amazon.kindle	14 (63%)
com.google...chrometophone	17 (81%)	com.game.CeramicDestroyer	4 (50%)	mobi.mgeek.TunnyBrowser	14 (78%)
org.connectbot	17 (100%)	com.google...googlevoice	4 (28%)	com.skype.raider	12 (50%)
com.facebook.katana	16 (18%)	com.linkedin.android	4 (40%)	com.farproc.wifi.analyzer	12 (60%)
com.agi.android.augmentedreality	14 (15%)	com.skype.raider	4 (17%)	com.weather.Weather	9 (53%)

Table 5: Variability of user opinions on behavior within major permission types. Each percentage indicates how often users flag it as objectionable in applications where it occurs. "Location" includes only API calls that can be performed with both location permissions.

Permission	Average	standard deviation	notes
camera	27%	30%	66% photo without preview; video 37%; preview, no photo 2.6%
read phone state	24%	13%	detecting phone calls 40%, IMEI 26%
location (either)	16%	9%	heavy resource usage 29%.
read sms	7%	7%	reading message contents 13%
internet	6%	7%	cookies 23%
write sms	5%	5%	

Users also had the option of indicating if they object to specific behaviors. Interestingly, the behaviors users object to do not overlap significantly with those exhibited by malware (see Table 6). For example, users seem most concerned with applications accessing their location, and less concerned about SMS behavior, even accounting for the fact that location behavior is more common. The fact that there is a marked difference between the set of unpopular behavior and the set associated with malware, in addition to the fact that there are many popular applications from well-established companies which are criticized heavily by users, suggests that there is a segment of users which are strongly concerned about privacy in general. Simply focusing on overtly malicious applications is not sufficient as there is a demand for transparency in the behavior of *all* applications.

Additionally, user opinions on different behavior types covered by the same permission varied greatly. This strongly suggests that permissions are providing information at the wrong granularity. On average, wherever any given behavior appears in an application profile, it is flagged as objectionable 11% of the time (standard deviation of 14%). Behavior falling under the Internet permission, for example, was flagged as objectionable 5.8% of the time on average, but the use of cookies specifically was flagged as objectionable 23% of the time. Some more data on common behaviors can be seen in Table 5.

Most notably, users only care about certain types of behavior covered by the "read phone state" permission. In particular, they object to anything that could be used to track them, or detect when they're making phone calls. For location, users care first about high resource usage, then about how fine-grained the data on their location is. Additionally, for an application to determine the user's distance to a given location, or to determine the direction in which they are traveling, is viewed as objectionably as using fine-grained location data. For camera-related behavior, users' opinions varied considerably between the four associated rules. In particular, using the camera to show a preview only was viewed as far less objectionable than the other behavior, which involved actually taking a picture. However, we have very few data points for some camera-related behavior. Finally, users are not particularly concerned about SMS messages, but they are slightly more concerned about SMS messages being read than messages being sent. The former has privacy implications, whereas the latter could cost them money and is common in malware. Perhaps privacy issues are viewed as a bigger threat than malware.

Additionally, there are 19 behavior types which correspond to none of the currently existing permissions. Of these, 6 never actually occur in any application we were given feedback on. These six all involve using obscure sensors. Users flagged the remainder as objectionable, on average, 5.5% of the time. Some, such as writing to the debug log or looking at phone orientation data, were never flagged as objectionable at all, and so perhaps not being covered by a permission makes sense in those cases. Being alerted when packages are installed, and using the accelerometer, however, were highly unpopular, both being flagged 9.5% of the time.

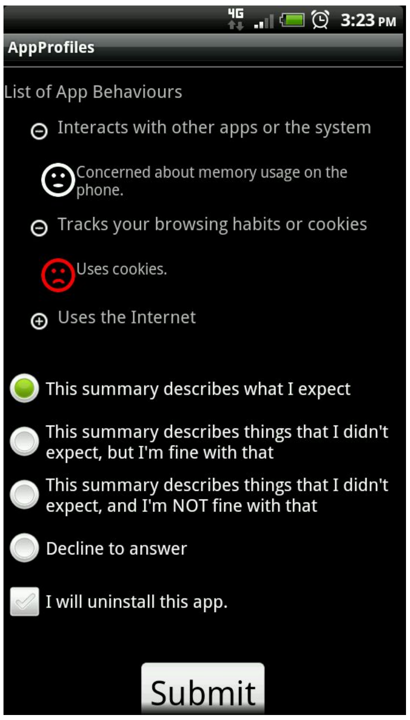
We later created a second feedback form with additional questions asking whether the users intend to uninstall the application, and whether the profiles had an impact on their view of the application. 71% of feedback items indicated that the profiles had changed the users' opinions, and 8.8% indicated that they would uninstall the application as a result of our profiles.

5.5 Lessons Learned

We have demonstrated that users find these profiles to be useful and informative, and that extracting this kind of data allows us to better understand behavior trends in the application market as a whole. We have also learned a great deal about user expectations of application behavior and the limitations of the permission system

Table 6: Comparison of the top ten most frequent behaviors observed in malware versus behaviors rated as objectionable by users.

Indicative of malware	disliked by users
SMS	**Location**
Send Text Message	Reads latitude or longitude
Copy SMS object	High resource use
Deprecated SMS manager	Misc. location use
Read contents of SMS	Use cached location
Privacy	**Privacy**
Get a unique id	Uses cookies
Get phone number	Get a unique id
Get carrier info	Detects phone calls
Internet	**Internet**
Send HTTP post	Check connection status
Miscellaneous	**Miscellaneous**
Use telephony manager	Take video
Triggered on boot completed	Uses the audio manager



List of App Behaviours

⊖ Interacts with other apps or the system

Concerned about memory usage on the phone.

⊖ Tracks your browsing habits or cookies

Uses cookies.

⊕ Uses the Internet

[✓ This summary describes what I expect]

This summary describes things that I didn't expect, but I'm fine with that

This summary describes things that I didn't expect, and I'm NOT fine with that

Decline to answer

I will uninstall this app.

Figure 5: Example of feedback form

as it currently stands, and so we offer a few suggestions as to how a permission system should be designed.

The permission system does not seem to be fine-grained enough to meet the needs of users. For example, the "Internet" permission may be too broad — for example, users care a great deal about the use of cookies, but not all kinds of network use. They care more about some types of location data than others, and care about reading phone IDs and phone numbers but not other aspects of the phone state. However, many of these distinctions cannot be made with the current permission system; permissions should be fine-grained enough to allow users to make these distinctions.

Furthermore, it seems valuable to differentiate between actions performed by users and actions performed in the background. For example, malware frequently sends SMS messages without the input of the user. It would be valuable to differentiate user-triggered actions from other actions in some cases and require different permissions accordingly, as has been suggested in previous work [32].

Finally, it seems that there are significant differences between third-party library code (such as that used by advertisements) and code written as part of a specific application. If users are greatly concerned about privacy, then they may be more concerned about libraries which might monitor them across applications, than an individual application. Differentiating between permissions unique to an application and those used by third-party libraries may be useful.

6. CONCLUSION

We have described a method for systematically detecting privacy-related application behavior in mobile systems where the most significant aspects of application behavior are mediated through a well-defined application framework. This method has two components; creating a knowledge base of API calls with privacy-relevant behavior, and using this knowledge base to produce behavior profiles for applications. We have demonstrated that it is a highly effective method of allowing both end users and researchers to better understand how applications behave. Finally, we have demonstrated that it is possible to create such profiles efficiently, given the almost 80,000 applications we have analyzed to date.

7. ACKNOWLEDGMENTS

This work is partly funded by NSF grants CNS-1059372, CNS-0964545, CNS-1050157, CNS-1039657 and Navy grant N00014-09-1-0705. We would like to thank Felt et. al., for making the Stowaway data set available, Enck. et al., for making ded available, and HP for providing us with a copy of HP Fortify SCA. We would like to thank Alex Halderman and Eric Wustrow for their valuable comments on the paper. We would also like to thank the anonymous reviewers whose comments helped improve the final version.

8. REFERENCES

[1] android-market-api — Android Market for all Developers! http://code.google.com/p/android-market-api/.

[2] Angry birds. https://play.google.com/store/apps/details?id=com.rovio.angrybirds.

[3] K. W. Y. Au, Y. F. Zho, Z. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *Proc. ACM Computer and Communications Security*, October 2012.

[4] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *Proc. ACM Computer and Communications Security*, October 2010.

[5] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-Based Malware Detection System for Android. In *Proc. ACM SPSM*, 2011.

[6] Carrier IQ Drama Continues. http://yro.slashdot.org/story/11/12/03/2112220/carrier-iq-drama-continues.

[7] Contagio Mobile. http://contagiominidump.blogspot.com/.

[8] A. Developers. Preparing for release. http://developer.android.com/tools/publishing/preparing.html.

[9] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proc. of USENIX Security Symposium*, August 2011.

[10] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, R. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. Operating Systems Design and Implementation*, October 2010.

[11] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proc. of USENIX Security Symposium*, August 2011.

[12] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *Proc. ACM Computer and Communications Security*, 2009.

[13] Facebook for android. https://play.google.com/store/apps/details?id=com.facebook.katana.

[14] Facebook Spies on Phone Users' Text Messages, Report Says. http://www.news.com.au/breaking-news/facebook-spies-on-phone-users-text-messages-report-says/story-e6frfku0-1226282017490.

[15] A. Felt, K. Greenwood, and D. Wagner. The Effectivenes of Application Permissions. In *Proc. USENIX Web Application Development*, 2011.

[16] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystified. In *Proc. ACM Computer and Communications Security*, 2011.

[17] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proc. SOUPS*, 2012.

[18] Hp fortify. https://www.fortify.com/.

[19] FortiGuard Threat Research and Response. http://www.fortiguard.com/.

[20] M. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *WiSec*, April 2012.

[21] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proc. Network and Distributed System Security Symposium*, 2012.

[22] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proc. Network and Distributed System Security Symposium*, February 2012.

[23] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. *J. Comput. Secur.*, 6(3):151–180, August 1998.

[24] P. Hornyaick, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proc. ACM Computer and Communications Security*, October 2011.

[25] X. Jiang. Mobile Security Alerts. http://www.csc.ncsu.edu/faculty/jiang/.

[26] Zero-Permission Android Applications. http://leviathansecurity.com/blog/archives/17-Zero-Permission-Android-Applications.html.

[27] List of Android Applications. http://nocrappyapps.com/media/app_ratings/androidappratings-current.csv.

[28] The Lookout Blog. http://blog.mylookout.com/.

[29] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proc. ACM Computer and Communications Security*, October 2012.

[30] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *ASIACCS*, 2010.

[31] Reddit is fun. https://play.google.com/store/apps/details?id=com.andrewshu.android.reddit.

[32] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems. In *Proc. IEEE Symposium on Security and Privacy*, May 2012.

[33] smali/baksmali. http://code.google.com/p/smali/.

[34] M. Szydlowski, M. Egele, C. Kruegel, and G. Vigna. Challenges for Dynamic Analysis of iOS Applications. In *iNetSec*, 2012.

[35] virustotal. https://www.virustotal.com/.

[36] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proc. Trust and Trustworthy Computing*, June 2011.