SAND2011-6211C

# AndroidLeaks: Automatically Detecting Potential Privacy Leaks In Android Applications on a Large Scale

Clint Gibler Jonathan Crussell Jeremy Erickson Hao Chen
{cdgibler, jcrussell, jericks}@ucdavis.edu, hchen@cs.ucdavis.edu

August 23, 2011

**Abstract**

As smartphones become more widespread and powerful, they store more sensitive data, which include not only the users' personal information but also the data collected via sensors throughout the day. When smartphone applications have access to this growing amount of sensitive information, they may leak it carelessly or maliciously.

The Android smartphone operating system provides a permissions-based security model that restricts an application's access to the user's private data. Each application statically declares the sensitive data and functionality that it requires in a manifest, which is presented to the user upon installation. However, the user does not know if the application is using the private data locally or sending it off the phone. To combat this problem, we present AndroidLeaks, a static analysis framework for automatically finding potential leaks of sensitive information in Android applications on a massive scale.

We evaluate the efficacy of AndroidLeaks on 24350 Android applications from several Android markets, from most of which we obtain every free application. We found 57299 potential privacy leaks in 7414 Android applications of private data including phone information, GPS location, WiFi data, and audio recorded with the microphone. AndroidLeaks examined these applications in 30 hours, which indicates that it is suitable for automatic analysis on a large scale.

## 1 Introduction

As smartphones have become ubiquitous, the focus of mobile computing has shifted from laptops to phones and tablets. Today, there are several competing mobile platforms, and as of March 3, 2011, Android has the highest market share of any smartphone operating system in the U.S. [6] Android provides the core smartphone experience, but much of a user's productivity is dependent on third-party applications. To this end, Android has numerous marketplaces at which users can obtain third-party applications. In contrast to the market policy for iOS, in which every application is reviewed before it can be posted [12], most Android markets are open for developers to post their applications directly, with no review process. This policy has been criticized for its potential vulnerability to malicious applications. Google instead allows the Android Market to self-regulate, with higher-rated applications more likely to show up in search results.

Android sandboxes each application from the rest of the system's resources in an effort to protect the user [2]. This attempts to ensure that one application cannot tamper with another application or the system as a whole. If an application needs to access a restricted resource, the developer must statically request permission to use that resource by declaring it in the application's manifest file. Then, when a user attempts to install the application, Android will warn the user that the application requires certain restricted resources (for instance, location data), and that by installing the application, she is granting permission for the application to use the specified resources. If the user declines to authorize the application, the application will not be installed.

However, statically requiring permissions does not inform the user how the resource will be used once granted. A maps application, for example, will require access to the Internet in order to download updated map tiles, route information and traffic reports. It will also require access to the phone's location in order to adjust the displayed map and give real-time directions. The application will send location data to the maps server in order to function, which is acceptable given the purpose of the application. However, if the application is ad-supported it may also leak location data to advertisers for targeted ads, which may compromise a user's privacy. Given the only information currently presented to users is a list of required permissions, a user will not be able to tell how the maps application is handling her location information.

To address this issue, we present AndroidLeaks, a static analysis framework designed to identify potential leaks of personal information in Android applications on a large scale. Leveraging WALA [5], a program analysis framework for Java source and byte code, we create a call graph of an application's code and then perform a reachability analysis to determine if sensitive information may be sent over the network. If there is a potential path, we use dataflow analysis to determine if private data reaches a network sink.

Our contributions in this paper are as follows:

- We have created a set of mappings between Android API methods and the permissions they require to execute. We use a subset of this mapping as the sources and sinks of private data for our dataflow analysis.
- Using this mapping we demonstrate the ability of our analysis to be a developer aid, automatically recovering the minimal set of permissions an application needs. We confirm the usefulness of this functionality based on observing applications with incorrectly specified permissions, including applications that misspell permissions or fail to include permissions required by API calls they use.
- We present AndroidLeaks, a static analysis framework which finds potential leaks of private information in Android applications. We evaluated AndroidLeaks on 24350 Android applications, finding potential privacy leaks involving uniquely identifying phone information, location data, WiFi data, and audio recorded with the microphone.
- We compare the prevalence of several popular ad libraries and the private data they leak.

## 2 Background

Android applications are primarily written in Java. Unlike standard Java applications, after being compiled into Java bytecode Android applications are converted into the Dalvik Executable (DEX) format. This conversion occurs because Android applications run in the Dalvik [4] virtual machine, rather than the Java virtual machine. Fortunately for our analysis, the conversion to DEX byte code retains enough information that the conversion is reversible in most cases. We were able to convert all but 15.99% of our Android applications from DEX into a JAR using the *dex2jar* tool [15].

Android applications are distributed in compressed packages called Android Packages (APKs). APKs contain everything that the application needs to run, including the code, icons, XML files specifying the UI, and application data. Android applications are available both through the official Android Market and other third-party markets. These alternative markets allow users freedom to select the source of their applications.

The official Android Market is primarily user regulated. The ratings of applications in the market are determined by the positive and negative votes of users. Higher ranked applications are shown first in the market and therefore are more likely to be discovered. Users can also share their experiences with an application by submitting a review. This can alert other users to avoid poorly behaving applications. Google is able to remove any application not only from the market, but also from users' phones directly, and has done so recently when users reported malicious applications [14, 18]. However, recent research [7] shows that many popular applications still leak their users' private data.

Android applications are composed of several standard components which are responsible for different parts of the application functionality. These components include: Activities, which control UI screens; Services, which are background processes for functionality not directly tied to the UI; BroadcastReceivers, which passively receive messages from the Android application framework; and ContentProviders, which provide CRUD operations[1] to application-managed data. In order to communicate and coordinate between components, Android provides a message routing system based on URIs. The sent messages are called Intents. Intents can tell the Android framework to start a new Service, switch to a different Activity, or to pass data to another component.

Each Android application contains an important XML file called a manifest [1]. The manifest file informs the Android framework of the application components and how to route Intents between components. It also declares the specific screen sizes handled, available hardware and most importantly for this work, the application's required permissions.

Android uses a permission scheme to restrict the actions of applications [2]. Each permission corresponds to protecting a type of sensitive data or specific OS functionality. For example, the INTERNET permission is required to initiate any network communications and READ_PHONE_STATE gives access to phone-specific information. Upon application installation, the user is presented with a list of required permissions. The user will be able to install the application only if she grants the application all the permissions. Without modifying the Android OS, there is currently no way to install applications with only a subset of the permissions they require. Additionally, Android does not allow any further restriction of the capabilities of a given application beyond the permission scheme. For example, one cannot limit the INTERNET permission to only certain URLs. This permission scheme provides a general idea of an application's capabilities, however, it does not show how an application uses the resources to which it has been allowed access.

## 3 Threat Model

In this work we consider a *privacy leak* to be any transfer of personal or phone-identifying information off of the phone. We do not attempt to distinguish personal data used by an application for user-expected application functionality from unintended or malicious use nor do we attempt to differentiate between benevolent and malicious leaks. Determining program intent is in general an

[1]Create, Read, Update, and Delete operations.

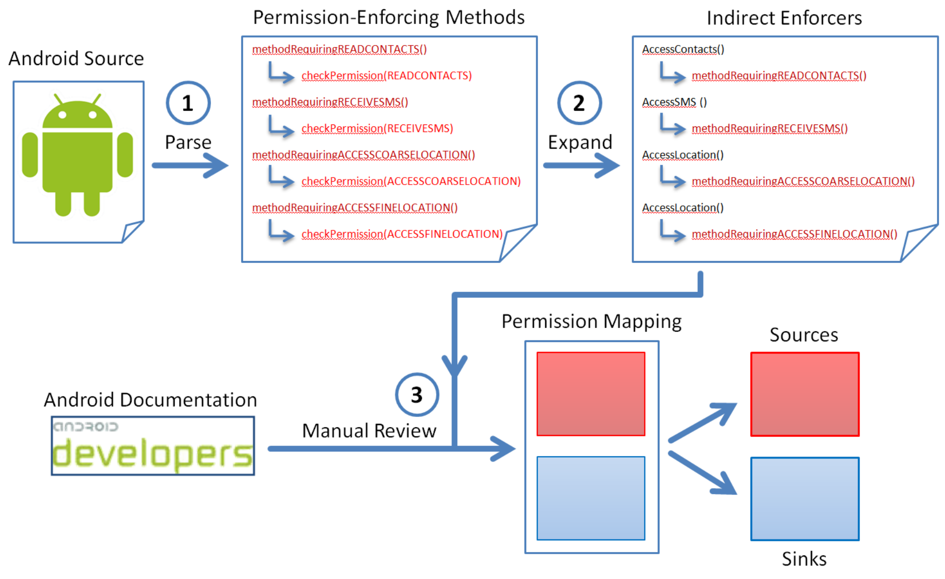

Android Source
1
Parse
Permission-Enforcing Methods
methodRequiringREADCONTACTS()
checkPermission(READCONTACTS)
methodRequiringRECEIVESMS()
checkPermission(RECEIVESMS)
methodRequiringACCESSCOARSELOCATION()
checkPermission(ACCESSCOARSELOCATION)
methodRequiringACCESSFINELOCATION()
checkPermission(ACCESSFINELOCATION)
2
Expand
Indirect Enforcers
AccessContacts()
methodRequiringREADCONTACTS()
AccessSMS ()
methodRequiringRECEIVESMS()
AccessLocation()
methodRequiringACCESSCOARSELOCATION()
AccessLocation()
methodRequiringACCESSFINELOCATION()
Android Documentation
android developers
3
Manual Review
Permission Mapping
Sources
Sinks

Figure 1: Creating a Mapping between API Methods and Permissions.

unsolvable problem, so we do not attempt to classify leaks as benign or malicious. Identifying personal data used for expected functionality requires understanding the purpose of the application as well as the intention of the developer during its creation, neither possible programmatically. Thus we classify transfer of personal information off of the phone as a privacy leak regardless of its use. Malware authors may maliciously leak private data, ad libraries may leak it for more targeted ads, applications may use it for their functionality- we attempt to address the general problem, tracking sensitive information flow on real applications at a large scale. We leave determining privacy leak intent to future work.

Our work focuses on Android applications leaking private data within the scope of the Android security model [2]. We are not concerned with vulnerabilities or bugs in Android OS code, the SDK, or the Dalvik VM that runs applications. For example, a Webkit[2] bug that causes a buffer overflow in the browser leading to arbitrary code execution is outside the scope of our work. Our trusted computing base is the Android OS, all third party libraries (not included in the APK), and the Dalvik VM.

We do not attempt to track private data specific to an application, such as saved preferences or files, since automatically determining which application data is private is very difficult. Finally, we do not attempt to find leaks enabled by the collaboration of applications, though there are no fundamental limitations to AndroidLeaks that prevent it from analyzing the potential collaboration of applications in future work.

Currently AndroidLeaks does not analyze native code. We do not believe this significantly affects our results as we found only 7% of Android applications include native code. Potentially, future Android malware could be nearly entirely written in native code to foil existing Java-based analysis tools. However, a malicious application can not hide its interaction with private data, as private data on Android may only be obtained by applications through Android's Java APIs. An application that accesses private data and immediately passes it to native code is a clear indicator

[2]Webkit is a rendering engine used by browsers such as Chrome and Safari.

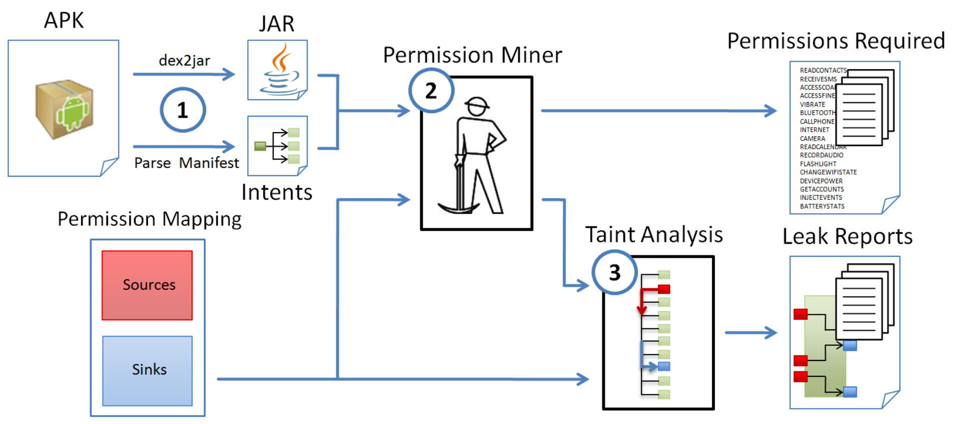


Figure 2: AndroidLeaks Analysis Process. 1. Preprocessing. 2. Recursive call stack generation to determine where permissions are required. 3. Dataflow analysis between sources and sinks.

for further analysis.

## 4 Methodology

In this section we discuss the architecture and implementation of AndroidLeaks. First we describe our process of creating our *permission mapping* — a mapping between Android API calls and permissions they require to execute. A subset of this mapping is used as the sources and sinks we include in our later dataflow analysis. By source, we mean any method that accesses personal data; for example, a uniquely identifying phone number or location data. We consider a sink to be any method which can transmit sensitive data off of the phone. In this paper, we focus on network connections. However, we have identified API methods for SMS and bluetooth sinks for inclusion in further work.

### 4.1 Permission Mapping

To determine if an application is leaking sensitive data, first one must define what should be considered sensitive data. Intuition and common sense can give a good starting point. However, in Android we can do much better since access to restricted resources is protected by permissions. Thus, if we can determine which API calls require a permission that protects sensitive data, it is likely that the methods are *sources* of private data.

Ideally this mapping between API methods and the permissions they require would be stated directly in the documentation for Android. This mapping would be useful for developers because it would help them better understand the permissions their application will require. Unfortunately, the Android documentation is incomplete, and only a partial mapping is provided.To address this issue, we attempt to automatically build this mapping by directly analyzing the Android framework source code. Figure 1 visualizes our process.

Intuitively, for a permission to protect certain API functionality, there must be points in the code where the permission is enforced. In manual analysis of the Android source, we found a number of helper functions that enforce a permission, such as *Context.enforcePermission(String, int, int)*, where the first parameter is the name of the permission. For every method in every class of the Android framework, we recursively determined the methods called by each method in the framework, building a call stack of the Android source, a process we call *mining.* Our miner will use all possible targets of virtual methods, erring on the side of completeness, rather than precision. If our mining encounters one of these enforcement methods, we inspect the value of the first parameter in order to determine the name of the permission being enforced. We then propagate the permission requirement to all the methods in the current call stack. After the permission mining is complete, we have a mapping between methods and the permissions they require. A subset of the methods in this mapping are API methods which are directly available to developers through the SDK.

To supplement our programmatic analysis, we manually reviewed the Android documentation to add mappings we may have missed. In particular, at some points in the Android framework, it may check, but not enforce a permission using a method such as *Context.checkPermission(String, int, int).* For each of these points in the code, we determined how the check was used and what method actually requires that permission and add it to our permission mapping before the mining process. Currently we have mappings between over 2000 methods and the permissions they require. To check the completeness of our mapping, we plan to collaborate with the group that worked on [9], which has also created a permission mapping but with dynamic testing.

Though this process gave us many mappings, it does not find permission checks that are implemented in native code and can not propagate permission requirements along edges connected by Intents or by IPC to a system process. While this may seem significant, we note that Android related control flow is outside of the scope of our current work and that we only found two permissions enforced outside of Java. The first of these two permissions is Internet, for which we manually added a very complete mapping. The second is Write External Storage, which is unimportant for our current work.

The primary focus in this paper is finding privacy leaks; however, our permission mapping contains method signatures for almost all permissions, not just the ones that access or can potentially leak sensitive data. This mapping could be used to aid developers in understanding more precisely which permissions different application functionality requires. Furthermore, our mapping could be used to automatically generate an application's required permissions, saving the developer time and assuring a minimal set of permissions. We describe our current effectiveness at automatically generating required permissions in Section 5.2.

## 4.2 Android Leaks

In this section we describe AndroidLeaks' analysis process. See Figure 2 for a visual representation. Before we attempt to find privacy leaks, we perform several preprocessing steps. First, we convert the Android application code (APK) from the DEX format to a jar using *dex2jar* [15]. This conversion is key to our analysis, as WALA can analyze Java byte code but not DEX byte code. Unfortunately, we found the *dex2jar* conversion fails on some JARs. We minimize the number of applications we cannot analyze by iteratively attempting to convert APKs with several versions of *dex2jar*, ranging from the last stable version to the latest branch in development. If those fail, we try converting the APK with *dedex* [**?**]. As AndroidLeaks does not rely on a given conversion tool, a better one may easily be swapped in at any time.

Using WALA, we then build a call graph of the application code and any included libraries. We iterate through the application classes and determine the application methods that call API methods which require permissions. We also keep track of which other application methods can call these application methods that require permissions, as reviewing the call stacks can give insight into the flow of the application's use of permissions. If the application contains a combination of permissions that could leak private data, such as *READ_PHONE_STATE* and *INTERNET*, we then perform dataflow analysis to determine if information from a source of private data reaches a network sink.

### 4.2.1 Taint Problem Setup

The three components of taint problems are sources, sinks and sanitizers. We rely on our permission mapping to categorize API methods requiring permissions relating to location, network state, phone state, and audio recording as sources.

We labeled all methods that require access to the Internet as sinks. However, our initial mapping contained very few mappings. We discovered that the Internet permission is enforced by the sandbox, which will cause any open socket command to fail if the Internet permission has not been granted. Since this permission is handled by native code, we were unable to automatically find many Internet permission mappings. A complete Internet mapping is very important, since it is the primary way to leak private data, so we manually went through the documentation for the *android.net*, *java.net* and *org.apache* packages and added undiscovered methods to our mapping.

Android has two categories of location data: coarse and fine. Coarse location data uses triangulation from the cellular network towers and nearby wireless networks to approximate a device's location, whereas fine location data uses the GPS module on the device itself. We do not differentiate between coarse and fine location data as we believe any leakage of location information to be important.

We do not include any sanitizers in our analysis for several reasons. Most importantly, we wanted to find paths where sensitive data is leaked off the phone regardless of if it has been processed in some way. Furthermore, we do not believe most applications will attempt to sanitize sensitive information they are sending to third parties. Lastly, recognizing application-specific data sanitization methods is difficult and not worth pursuing at this stage of our work.

### 4.2.2 Taint Analysis

First, we use WALA to construct a context-sensitive System Dependence Graph (SDG) with no heap information. The SDG is a graph that is comprised of many Program Dependence Graphs (PDGs), which show the dependencies within a single method. By adding special nodes between method calls, the SDG expresses interprocedural dependencies. Since context-sensitive pointer analysis is resource intensive, we use chose to use a context-insensitive overlay to show heap dependencies in the SDG. Using the SDG, we compute forward slices for the return value of each source method we identify in the application. We use the return value because all the sources we have identified return sensitive data through the return value and not through other means like side-effects on the parameters. We then analyze the slice to determine if any parameters to sink methods are tainted, meaning that they are data dependent on the source method. If such a dependency exists, then private data is most likely being leaked and we record it.

Unfortunately, WALA's built-in SDG and forward slicing algorithms alone are not sufficient to do taint tracking. In order to accomplish this, we added the following functionality:

**Handling Callbacks** Most sources are API methods, however, callbacks are used extensively in Android and there are some that will be called with private data as a parameter. For example, location information can be accessed either directly by asking the LocationManager for the last known location or by registering with the LocationManager as a listener. For the latter, the LocationManager provides regular updates of the current location to the registered listener. For API methods labeled as sources, we were able to taint the return values of these methods, however, for callbacks this approach does not work since neither the return value of the callback nor the return value of the registration is tainted. Instead, we automatically identified calls to the register listener method while mining the application code and then inspected the parameters to determine the type of the listener. We then tainted the parameters of the callback method for the listener's class. This approach allows us to compute forward slices for both types of access in the same way.

**Taint-Aware Slicing** Rather than modify WALA internally as done in [17] to achieve taint-aware slicing, we decided to analyze the computed slices and compute new statements from which to slice. We implemented the following logic to compute these new statements:

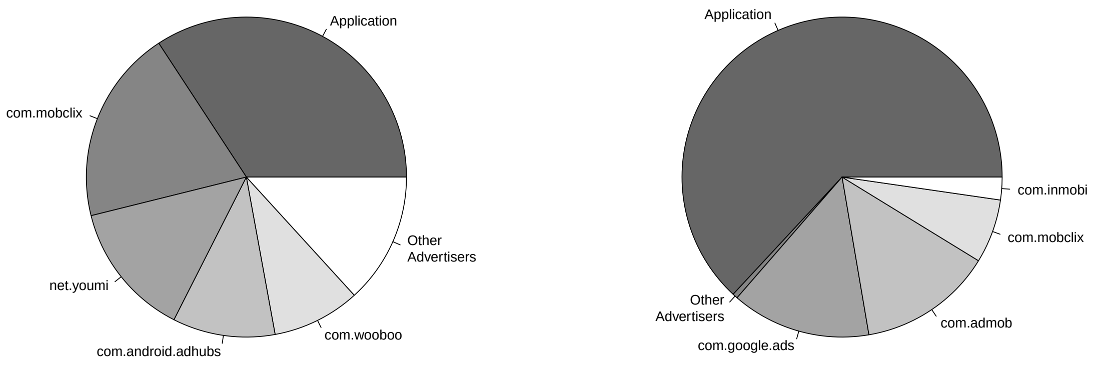
1. Taint all objects whose constructor parameters are tainted data.
2. Taint entire collections if any tainted object is added to them.
3. Taint whole objects which have tainted data stored inside them.

By applying these propagation rules to the slice computed for the source method, we create a set of statements that are tainted but would not be included in the original slice because the original slice shows statements that are data dependent which is only part of how taint propagates. We then compute forward slices for each of these statements and all others derived in the same manner from subsequent slices until we encounter a sink method or run out of statements from which to slice.

Of our propagation rules, the third rule causes the most propagation of taint and is therefore the largest source of false positives. An example of a false positive we saw occurred when an Activity became tainted. Activities are responsible for coordinating all the functionality for a UI screen, so if one part of the UI interacts with private data and another sends data to the network, AndroidLeaks may report a potential leak. For example, if an Activity has an instance variable that stores the current location, taint will be propagated to the entire Activity object. If the Activity also has an ad window that pulls advertisements from the Internet, then there is a potential leak. As we intend the results of AndroidLeaks to be manually reviewed, we err on the side of over tainting so that we find when this tainting is correct, at the expense of potential false positives. Preventing over tainting while properly tracing information flow is a difficult problem with static analysis, especially when complex objects handle both tainted and untainted data. We note that [17], on which our taint analysis was based, also has high false positives in certain cases.

## 5 Evaluation

We evaluated AndroidLeaks on a body of 28983 unique Android applications. The official Android Market [11] has many free applications but Google has created mechanisms to discourage automated crawling. Fortunately, the application distribution model of Android applications works in our favor — there are many third-party Android market sites. We obtained applications from several American and Chinese markets, including SlideMe [16] and GoApk [3]. We found that many applications, which we identify by their SHA1 hashes, are present in multiple markets.



(a) Phone Identification (b) Location

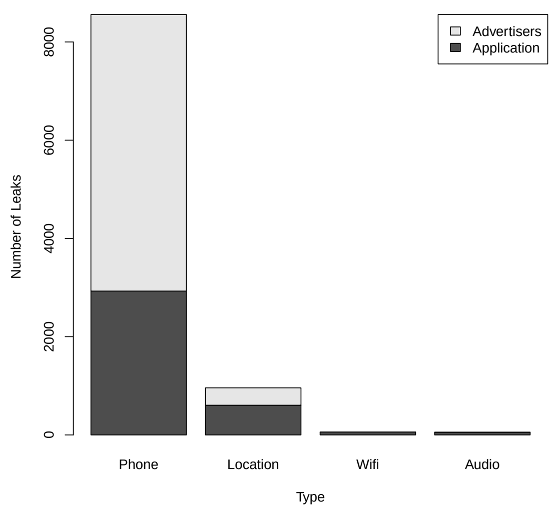
Figure 3: Leaks by Source Location

Out of these 28983 applications we were unable to analyze 4633 due to invalid bytecodes in the *dex2jar* converted APKs. There were also 1626 applications which required no permissions. These apps do not have the ability to gain access to sensitive data nor leak it so we exclude them from the analysis described in this section. We found potential privacy leaks in 7414 of the remaining 24350 apps.

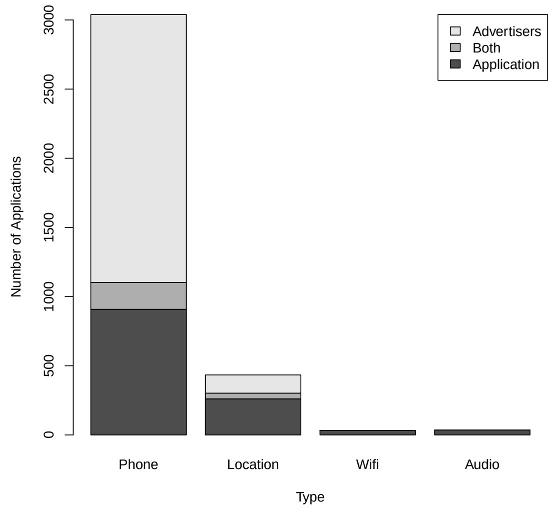
Running AndroidLeaks on one server-grade computer we were able to analyze all 24350 applications in 30 hours- over 800 APKs per hour. Collectively we processed over 531249 unique Java classes.

We chose to focus on 4 types of privacy leaks: uniquely identifying phone information, location data, WiFi state and recorded audio. Examples of uniquely identifying phone information include the unique device ID (IMEI for GSM phones, MEID or ESN for CDMA phones) and the subscriber ID (IMSI for GSM phones). For location data, AndroidLeaks tracks accesses to both "coarse" and "fine" GPS data. WiFi state information includes the SSID and BSSID of the current access point as well as the MAC address of the phone's WiFi adapter. Though information about the WiFi networks seen by a phone may not seem sensitive, correlating this with a broad knowledge of the location of wireless networks can yield a device's specific location. In fact, Android phones already offer the option in the phone's "Location and Security" settings to use nearby wireless networks to determine the phone's location. Finally, we include audio recorded with the phone's microphone.

The importance of a given privacy leak varies depending on the sensitivity of the data being leaked and the privacy concerns of the user. We design AndroidLeaks to find leaks ranging in sensitivity to allow users of AndroidLeaks to focus on findings at their desired level of privacy.



(a) Breakdown of App vs Ad Code Leaks by Type



(b) Apps that contain Leaks in Ad Code, App Code or Both

Figure 4: Leaks by Type

## 5.1 Potential Privacy Leaks Found

We found a total of 57299 leaks in 7414 Android applications. 7870 of these are unique leaks, varying by source, sink or code location. 36388 were leaks found in ad code, which comprises 63.51% of the total leaks found. In Figure 3 we show the source of leaks of phone and location data, divided into leaks found in application code and ad libraries. We do not include pie charts for WiFi and record audio leaks because all of these leaks were found in application code. Interestingly, the majority of phone-related privacy leaks were found in ad code while application code contained the most location data leaks. Ad libraries were responsible for 66% of the total phone data-related leaks with the top four ad libraries accounting for 53%. Application code contained 63% of the privacy leaks and the top four ad libraries covering 36%. Figure 4a shows a breakdown of the leaks found by the type of leak and its source. Figure 4b displays the number of applications we found containing each type of leak, organized by who is leaking the information. We found that in most cases where phone identifying information is leaked, the advertising library is solely responsible.

### 5.1.1 Verification

Due to the large number of APKs analyzed and leaks found, it is fundamentally difficult to manually verify the correctness of a large percentage of our results due to time constraints. We chose to focus on verifying leaks found in ad code to gain a maximum amount of insight into the accuracy of our results. Ad code is almost always a third-party library that is included with application code by the developer, and a given version of an ad library should be the same between applications. Therefore, by confirming an ad leak as a true or false positive we can reuse that result for all occurrences of that same leak. We identify leaks to be identical by a 3-tuple: source method, method the source method is called from, and the sink method. Thus we focused on verifying the most common unique leaks to determine our accuracy on the largest number of leaks.

We manually traced 48 leaks in various versions of the Mobclix, adHUBS, Millennial Media, and Mobclick ad libraries to assess the accuracy of AndroidLeaks' results. Of these, we were able to verify 24 to be valid leaks in ad code. The false positives tended to occur most commonly in applications that contained many ad libraries in addition to the one in which we were analyzing. As multiple ad libraries may populate UI components on the same screen, our analysis may conservatively say that it's possible for sensitive data accessed by one ad library to propagate to its containing Activity or other ad libraries that share the same Activity.

The 24 leaks described above are collectively repeated 4588 times and occur in 1983 unique applications. Therefore at least a third of the potential leaks AndroidLeaks discovered are confirmed true positives and at least half of the total reported leaky APKs have confirmed leaks.

We also verified a small random set of 20 applications containing each leak type in application code to confirm AndroidLeaks is successful at finding leaks in application code as well. Several of the microphone leaks we verified turned out to be in IP camera applications, such as "SuperCam" or "IP Cam Viewer Lite."

It is important to reiterate that AndroidLeaks reports potential privacy leaks but cannot automatically verify its results. Manual verification by the application developer or a security researcher is required to determine the veracity of the findings. To ease this process, AndroidLeaks specifies the containing class and method as well as each leak's source and sink. AndroidLeaks can guide and focus a manual reviewer's efforts to allow her to analyze many times more applications than she could manually.

### 5.1.2 Ad Libraries

Nearly every ad library we looked at leaked phone data and, if possible, location information as well. We hypothesize that nearly any access of sensitive data inside ad code will end up being leaked, as ad libraries provide no separate application functionality which requires accessing such information.

As an application developer, knowledge of the types of private information an ad library may leak is valuable. One may use this knowledge to select the ad library that best respects the privacy of users and possibly warn users of potential uses of private information by the advertising library.

One solution is to watch an application that uses a given ad library using dynamic analysis, such as TaintDroid. However, one runs into limitations of dynamic analysis, such as difficulty in achieving high code coverage. Manually driving applications through all code paths is infeasible at the rate new Android applications are being published, between 7,500 and 22,500 per month according to [?]. But even with maximum possible code coverage using dynamic taint analysis, there are further challenges on Android. Many ad libraries we examined check if the application they were bundled with has a given permission, oftentimes the ability to access location data. Using this information, they could localize ads, potentially increasing ad revenue by improving click through rates. However, there is nothing preventing ad libraries from checking if they have access to any number of types of sensitive information and attempting to leak them only if they are able. A dynamic analysis approach could watch many applications with a malicious advertising library and never see this functionality if none of the applications declared the relevant permissions. Using our static analysis approach we do not have this limitation and would be able to find these leaks regardless of the permissions required by the application being analyzed.

| Leak Type | # Leaks | % of all Leaks | # apps with leak | % apps with leak |
|---|---|---|---|---|
| Phone | 53281 | 92.99% | 6912 | 28.39% |
| Location | 3405 | 5.94% | 969 | 3.98% |
| WiFi | 266 | 0.46% | 79 | 0.32% |
| Record Audio | 347 | 0.61% | 115 | 0.47% |

Table 1: Breakdown of Leaks by Type

| | Type of Leak | | | | | |
|---|---|---|---|---|---|---|
| Ad Library | Phone | Location | Wifi | Microphone | # apps using | % apps using |
| Mobclix | ✓ | O | X | X | 7861 | 32.28% |
| Youmi | ✓ | O | X | X | 6423 | 26.38% |
| Wooboo | ✓ | O | X | X | 5684 | 23.34% |
| adHUBS | ✓ | O | X | X | 3285 | 13.49% |

Table 2: Ad Library Leaks by Type. ✓: found by our analysis, 0: missed by analysis but found manually, X: not found by either

Ad libraries tend to be distributed to developers in a precompiled format, so it is not easy for an application developer to determine the information the ad library uses for user analytics. This is important for developers that include ad libraries in highly sensitive applications because the developer is ultimately responsible for any information leaked by libraries they choose to include. Additionally, a developer wanting to use an ad library is forced to use the ad library as it comes, with no option to remove features or modify the code. Since there is no mechanism in Android that allows one to restrict the capabilities of a specific portion of code within an application — all ad libraries have privilege equal to the application with which they are packaged. We note that a need for sand-boxing a subset of an application's code is not an issue specific to Android; it is an open issue for many languages and platforms. However, the issue is especially relevant on mobile platforms because applications commonly include unverified third-party code to add additional features, such as ads.

## 5.2 Discovering Required Permissions

The Permission Mining step of our analysis can be used by Android developers to automatically generate the minimal set of permissions their application requires. Of the permissions we focused on for detecting privacy leaks,[3] AndroidLeaks precisely recovered the exact subset of those permissions required for 66% of them. As AndroidLeaks does not currently take into account API calls that occur after tests for the existence of a permission, it currently finds applications containing ad libraries that attempt to use location information as requiring location permissions even if the application's manifest does not allow the ad library to do so. Discounting cases involving location information used only in ad code, we precisely recover the permissions of 89% of applications.

Thus, at least for a subset of the permissions, AndroidLeaks is effective at recovering an application's required permissions. We plan to improve our mapping in future work to increase this

[3]INTERNET, READ_PHONE_STATE, ACCESS_[COARSE|FINE]_LOCATION and RECORD_AUDIO

accuracy and to precisely determine the remaining Android-defined permissions.

**Likely Developer Permission Errors** Android gives developers the flexibility to define permissions specific to their application to allow them to share functionality with other applications in a mediated fashion. However, as developers must manually specify the permissions their application needs and are not restricted to a predetermined set, there is room for developer error. While it's difficult to definitively say that a permission was incorrectly specified without manually reviewing the code, as it could be an application-specific permission, we found a number of permissions that appear to be typographical errors. [4] Out of 28983 applications, there were 669 unique permissions declared. Based solely on the permission names, we estimate at least 125 of these to be developer errors, based on their appearance of being misspellings or misinterpretations of existing Android permissions. We also found that 82 applications declaring no permissions contained API calls that require permissions. For example, several of these applications included an ad library. Without the INTERNET permission, ad libraries cannot obtain ads to display to users, rendering them useless. While these developer errors occur in only a small percentage of applications, they lend credence to the idea that there may be more ways in which developers incorrectly specify permissions and that automatically generating an application's permissions is valuable.

## 5.3 Miscellaneous Findings

**Unique Android Static Analysis Issues** During the course of our analysis, we found several issues unique to Android that impacted our false positive and false negative rate. A common programming construct in ad libraries is to check if the currently running application has a certain permission before executing functionality that requires this permission. Many ad libraries do this to serve localized ads to users if the application has access to location data. An analysis that does not take this into account would find all such libraries as requiring access to location data and would possibly find leaks involving location data when in reality neither are valid because the application does not have access to location data.

**Native Code** Native code is outside the scope of our analysis, however, it is interesting to see how many applications use native code. The use of native code is discouraged by Android as it increases complexity and may not always result in performance improvements. Additionally, all Android APIs are accessible to developers at the Java layer, so the native layer provides no extra functionality. We found that 1,988 out of 28983 applications (7%) have at least one native code file included in their APK. Of the total 3,902 shared objects in APKs, a majority (2,014, 52%) of them were not stripped. This is interesting because stripping has long been used to reduce the size of shared libraries and to make them more difficult to reverse engineer, however, a majority of the applications we downloaded contained unstripped shared objects. This may be a result of developers using C/C++ who aren't familiar with creating libraries.

[4] "WRITE_EXTERNAL_STOR**EA**GE"
"ACCESS_CO**U**RSE_LOCATION"
"and**oir**d.permission.ACCESS_COARSE_LOCATION"

## 6 Limitations

### 6.1 Approach Limitations

There are several inherent limitations to static analysis. Tradeoffs are often made between speed, precision, and false positives. We chose to have AndroidLeaks err on the side of false positives rather than false negatives as we intend for results to be manually verified. Thus, while it could assist, we do not intend for AndroidLeaks alone to be an Android market "gatekeeper" that applications must pass to be published or a definitive reporter of leaking applications.

While a dynamic approach would have high precision due to the fact that privacy leaks are directly observed at run-time, having high code coverage is a challenging problem. Dynamic analysis tools [7] tend to be manually driven, which does not scale to tens of thousands of Android applications, as was our goal. Combining AndroidLeaks with a dynamic approach could have great potential, as AndroidLeaks could quickly scan many applications and determine candidates for further analysis. We leave combining AndroidLeaks with a dynamic analysis approach open to future work.

### 6.2 Implementation Limitations

**Incomplete permission mapping** Our mapping between API methods and permissions has a high coverage of the Android API but is potentially incomplete. Without testing every single Android API call and every native library function in several executing environments, it's difficult to tell if we are missing or have extra methods in our mapping. The Android OS has been evolving at a rapid rate, releasing a new version every few months. Maintaining a complete mapping in such a rapidly changing environment adds further difficulties. Our results demonstrate the usefulness and efficacy of our current permission mapping, though possibly incomplete, and we leave refining our mapping to future work.

**Android-specific control and data flows** AndroidLeaks does not yet analyze Android-specific control and data flows. This includes Intents, which are used for communication between Android and application components, and Content Providers, which provide access to database like structures managed by other components.

**Analysis dependencies** As mentioned in Section 5, we are unable to run our analysis on a portion of applications as a result of invalid bytecodes in the *dex2jar* converted APKs. We rely on both *dex2jar* and WALA working for us to analyze an application. Though it is possible for a malicious developer to purposefully create an APK that *dex2jar* incorrectly converts, we do not believe this is an important threat as there is no fundamental technical obstacle to creating a more reliable *dex2jar*. Additionally, our analysis does not inherently rely on *dex2jar*, another tool that more effectively converts the DEX format to Java byte code could easily be swapped in such as *ded* [8].

## 7 Future Work

**Android-specific control and data flow** As described in Section 6, there are several unique ways execution may flow in Android that we plan to handle in the future. Using Intents, Android components can interact with each other, either explicitly by name or implicitly by action or data. Both types of Intents are more complicated to analyze than standard Java control and data flow.

In the former case, we would need to determine the possible runtime values of the arguments in Intent passing and in the latter case we would need to build up a model of both the application's configured environment and potentially the other installed applications on the phone to know what would be called.

**Permission mapping** The permission mapping is a very important part of our work and its precision and completeness directly affect our results, creating both false positives and negatives. While our current mapping is sufficient for the scope of our current work, it will need to be improved moving forward. Once we have Android-specific control flow integrated into our analysis, we should be able to drastically improve the mapping. We plan to collaborate with [9] to improve the precision and completeness of our mapping.

## 8 Related Work

Chaudhuri et al. present a methodology for static analysis of Android applications to help identify privacy violations in Android with SCanDroid [10]. They used WALA to analyze the source code of applications, rather than Java byte code as we do. While their paper described mechanisms to handle Android specific control flow paths such as Intents which our work does not yet handle, their analysis was not tested on real Android applications.

Egele et al. perform similar analyses with their tool PiOS [13], a static analysis tool for detecting privacy leaks in iOS applications. AndroidLeaks and PiOS both found privacy leaks related to device ID, location and phone number. PiOS additionally considered the address book, browser history and photos while we consider several other types of phone data, WiFi data and audio recorded with the microphone. PiOS ignored leaks in ad libraries, claiming that they always leak, while one of the focuses of our work is giving developers insights into the behavior of ad libraries. To our knowledge, PiOS presented the largest public static analysis of smartphone applications before this paper, analyzing 1,400 iOS applications whereas we analyzed over 24000.

In comparison to AndroidLeaks's static analysis approach, TaintDroid [7] detects privacy leaks using dynamic taint tracking. Enck et al. built a modified Android operating system to add taint tracking information to data from privacy-sensitive sources. They track private data as it propagates through applications during execution. If private data is leaked from the phone, the taint tracker records the event in a log which can be audited by the user. Many of the differences between AndroidLeaks and TaintDroid are fundamental differences between static and dynamic analysis. Static analysis has better code coverage and is faster at the cost of have a higher false positive rate. One benefit of AndroidLeaks over the implementation of TaintDroid is that AndroidLeaks is entirely automated, while TaintDroid requires manual user interaction to trigger data leaks. We believe that AndroidLeaks and TaintDroid are in fact complementary approaches, AndroidLeaks can be used to quickly eliminate applications from consideration for dynamic testing while flagging areas to test on applications that are not eliminated.

Zho et al. presented a patch to the Android operating system that would allow users to selectively grant permissions to applications [19]. Their patch gives users the ability to revoke access to, falsify, or anonymize private data. While this is an effective way to limit permissions granted to applications, it requires flashing the phone's ROM, which voids most phone warranties and is too technical for many users.

Enck et al. [8] created *ded*, a tool that decompiles DEX to Java source code. They used *ded* to convert 1,100 free Android applications to Java source code that they then analyzed with a commercial static analysis tool. Because they used a commercial tool but never described its analysis algorithms, it's difficult to compare the merit of our analyses directly. From their preliminary results, we can note that Androidleaks is faster and therefore can run on a much larger scale. While just *ded*'s decompilation took approximately 20 days on 1,100 applications, our conversion and analysis time for 24000 applications was approximately 30 hours. Their analysis time was not specified.

Felt et al. investigated permission usage in 940 Android applications using their tool STOWAWAY [9]. In order to determine the API method to permissions mapping, they generated unit tests for each method in the Android API. They then executed these tests and observed if the execution caused a permission check. This dynamic approach is very precise, however, may be incomplete because their automated test construction may not have called API methods with arguments that cause the method to perform a permission check. Combining their mapping with our statically generated one could produce a very complete and precise mapping.

## 9 Conclusion

Android users need a way to determine if applications are leaking their personal information. Whereas Apple's App Store incorporates a review and approval process, Android markets relies on user regulation and reviews. Android uses a permissions model that limits applications' access to restricted resources. Our primary goal was to analyze privacy violations in Android applications. Along the way, we identified a mapping between API methods and the permissions they require, created a tool to discover the permissions an application requires, accumulated a database of almost 29000 Android applications and detected over 9000 potential privacy leaks in over 7400 applications.

## References

[1] Android developer reference. http://d.android.com/.

[2] Android security and permissions. http://d.android.com/guide/topics/security/security.html.

[3] Go Apk. Go apk. http://market.goapk.com.

[4] Dan Bornstein. Dalvik vm internals, 2008. Accessed March 18, 2011. http://goo.gl/knN9n.

[5] IBM T.J. Watson Research Center. T.j. watson libraries for analysis (wala), March 2011.

[6] The Nielsen Company. Who is winning the u.s. smartphone battle? Accessed March 17, 2011, http://blog.nielsen.com/nielsenwire/online_mobile/who-is-winning-the-u-s-smartphone-battle.

[7] William Enck, Peter Gilbert, Byung-Gon Chun, Landom P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. *OSDI*, 2010. http://appanalysis.org/tdroid10.pdf.

[8] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. *To Appear in Usenix Security*, 2011. http://www.enck.org/pubs/enck-sec11.pdf.

[9] Adrienne Porter Felt, Ericka Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. *To Appear in CCS*, 2011. http://www.cs.berkeley.edu/ afelt/android_permissions.pdf.

[10] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications. http://www.cs.umd.edu/ avik/papers/scandroidascaa.pdf.

[11] Google. Android market. http://market.android.com.

[12] Apple Inc. App store review guidelines. http://developer.apple.com/appstore/guidelines.html.

[13] Engin Kirda Manuel Egele, Christopher Kruegel and Giovanni Vigna. Pios: Detecting privacy leaks in ios applications. *NDSS'11.* http://www.iseclab.org/papers/egele-ndss11.pdf.

[14] Peter Pachal. Google removes 21 malware apps from android market. March 2011. Accessed March 18, 2011. http://www.pcmag.com/article2/0,2817,2381252,00.asp.

[15] pxb1988. dex2jar: A tool for converting android's .dex format to java's .class format. https://code.google.com/p/dex2jar/.

[16] SlideMe. Slideme: Android community and application marketplace. http://slideme.org/.

[17] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. Taj: Effective taint analysis of web application. In PLDI '09: Programming language design and implementation, pages 87-97. ACM, 2009.

[18] Sara Yin. 'most sophisticated' android trojan surfaces in china. December 2010. Accessed March 18, 2011. http://www.pcmag.com/article2/0,2817,2374926,00.asp.

[19] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming information-stealing smartphone applications (on android). *TRUST*, 2011. http://www.csc.ncsu.edu/faculty/jiang/pubs/TRUST11.pdf.