SAND2011-5590C

AndroidLeaks: Detecting Privacy Leaks In Android Applications

July 29, 2011

Abstract

As smartphones increase in prevalence and functionality, they have become responsible for a greater and greater amount of personal information. The information smartphones contain is arguably more personal than the data stored on personal computers because smartphones stay with individuals throughout the day and have access to a variety of sensor data not available on personal computers. Developers of smartphone applications have access to this growing amount of personal information, however, they may not handle it properly, or they may leak it maliciously.

The Android smartphone operating system provides a
permissions-based security model which restricts application's access to user's private data. Each application statically declares its requested permissions in a manifest file which is presented to the user upon application installation. However, the user does not know if the application is using the private locally or sending it to some third party. To combat this problem, we present AndroidLeaks, a static analysis framework for finding leaks of personal information in Android applications.

To evaluate the efficacy of AndroidLeaks on real world Android applications, we obtained over 23,000 Android applications from several Android markets. We found 9,631 potential privacy leaks in 3,258 Android applications of private data including phone information, GPS location, WiFi data, and audio recorded with the microphone.

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 leaks of personal information and privacy violations in Android applications on a large scale. Leveraging WALA[5], a program analysis framework for Java source and byte code, we create a callgraph of the application 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 perform dataflow analysis to determine if private data reaches a network sink.

Other projects, such as TISSA[19], have worked on allowing users more control over access to private data on a per application basis. However, taking advantage of their approach requires the user to flash a custom version of the Android Operating System. This currently prevents widespread adoption because there are barriers to doing this, such as voided warranties and lack of technical knowledge.

AndroidLeaks has several advantages over related work in privacy leak detection. By using static analysis techniques, we are able to cover the entire code base, identifying paths that may not be uncovered using dynamic analysis, as dynamic analysis may not be able to trigger all execution paths in the application. As AndroidLeaks does not require running applications, we are able to analyze many Android applications in a short period of time- 18,089 applications in under 26 hours, or almost 700 applications per hour. While several other tools exist to find privacy violations in Android applications[7, 10], to the best of our knowledge, none have automatically analyze applications on a large scale.

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 of private data and the network sinks we use to detect privacy leaks.

- 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 published applications with incorrectly specified permissions, including misspellings of Android-defined permissions.

- We present AndroidLeaks, a static analysis framework which finds leaks of private information in Android applications. We evaluated AndroidLeaks on 23,838 Android applications, which is to our knowledge the largest known evaluation of of mobile applications. We found potential privacy leaks involving uniquely identifying phone information, location data, WiFi data, and audio recorded with the microphone in 3,258 Android applications.

- We compare the permissions used and data leaked in several popular ad libraries. We manually verified a number of ad library leaks to help developers pick the most privacy-respecting ad libraries.

2 Background

Android applications run in a virtual machine called Dalvik [4]. A large portion of the Android framework and the applications themselves, are initially coded in Java, then compiled into Java bytecode before being converted into the Dalvik Executable (DEX) format. Fortunately for our analysis, the final conversion to DEX byte code retains enough information that the conversion is reversible in most cases 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 the application if it behaves poorly. 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, to switch to a different Activity, and 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

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

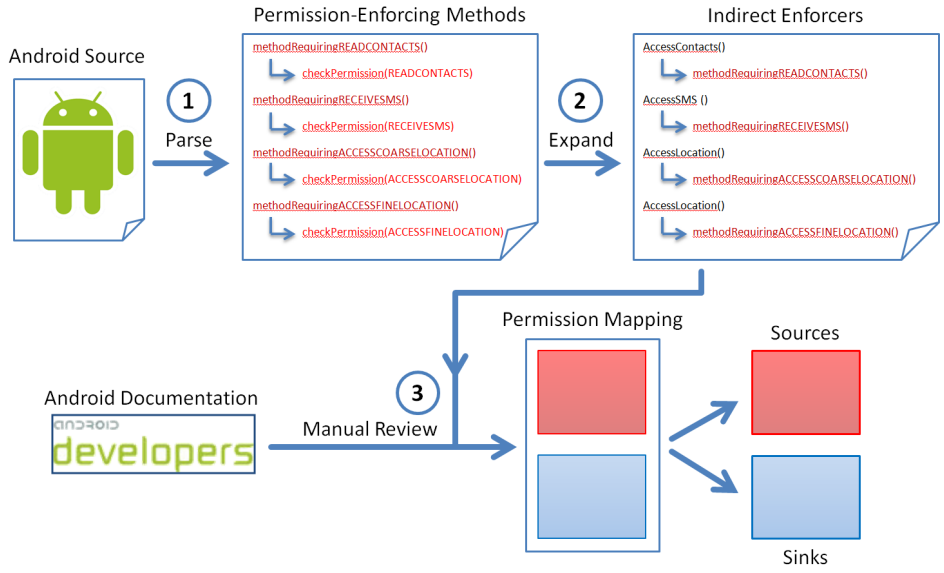


Figure 1: Creating a Mapping between API Methods and Permissions.

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 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 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.

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

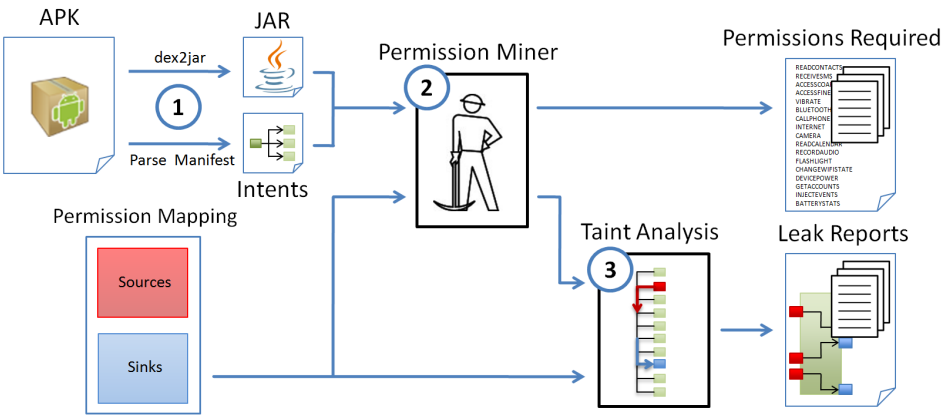


Figure 2: AndroidLeaks Analysis Process.
1. Preprocessing. 2. Recursive call stack generation to determine where permissions are required. 3. Dataflow analysis between sources and sinks.

Our goal is to determine if applications are using the permissions they are granted to leak sensitive data off the phone. Examples of this include sending the phone's unique ID number or location data to an user-analytics firm or to the application developer.

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. There are no fundamental limitations to AndroidLeaks that prevent it from analyzing the potential collaboration of applications,

Currently AndroidLeaks does not analyze native code. We do not believe this significantly affects our results as we found only 6% 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 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 what permissions their application will require. Unfortunately, the documentation is incomplete, and frequently will omit this mapping. 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 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*. If our mining encounters one of these enforcement methods, we inspect the value of the first parameter, which is always a constant in the Android framework, 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. The same propagation routine is done if we encounter a method already in our permission mapping. After mining is complete, we will 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. We note that this mapping includes both API methods and internal framework methods and that it is important to include internal framework methods in the mapping because they are accessible through reflection.

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. 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 our AndroidLeaks 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.

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 app methods can call these app methods that require permissions, as reviewing the callstacks 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 ever reaches a sink.

4.2.1 Taint Problem Setup

The three components of most taint problems are sources, sinks and sanitizers. In our setup, we rely on the permission mapping we built between API calls and the permissions they require to categorize permissions relating to location, network state, phone state, and audio recording as sources.

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 for two reasons. First, when we created our permission mapping, we discovered that methods that require ACCESS_COARSE_LOCATION will accept ACCESS_FINE_LOCATION instead. Second, because in practical use, using wireless networks can allow a coarse location fix to get as precise as 50 meters or less. We believe this to be almost as sensitive as fine location data.

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.

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) and then add a context-insensitive heap dependency overlay. Using the resulting SDG, we compute forward slices for the return value of each source method we identify in the application. 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 used the following approaches:

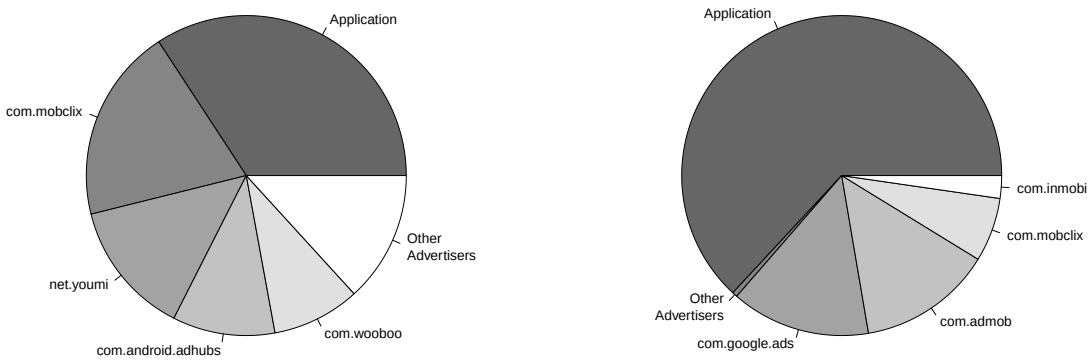
Handling Callbacks Most sources are API methods, however, callbacks are used extensively in Android and there are some that will be called with private data as a parameter. For example, location information can be accessed either directly by asking the LocationManager for the last known location or by registering with the LocationManager as a listener. If 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 identified calls to the register listener method 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], 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 are not be included in the original slice. 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 often occurred when Activities became tainted. Since Activities are responsible for coordinating all the functionality for a UI screen, if any part of that screen uses the Internet, there is a potential leak. While it may seem like this rule should be removed in order to prevent such false positives, it's important to note that without it, our taint analysis would likely find no leaks. Preventing over tainting while properly tracing information flow is a difficult problem with static analysis, especially when objects handle both tainted and non-tainted. We note that [17], on which our taint analysis was based, also has high false positives in certain cases.



(a) Phone Identification

(b) Location

Figure 3: Leaks by Source Location

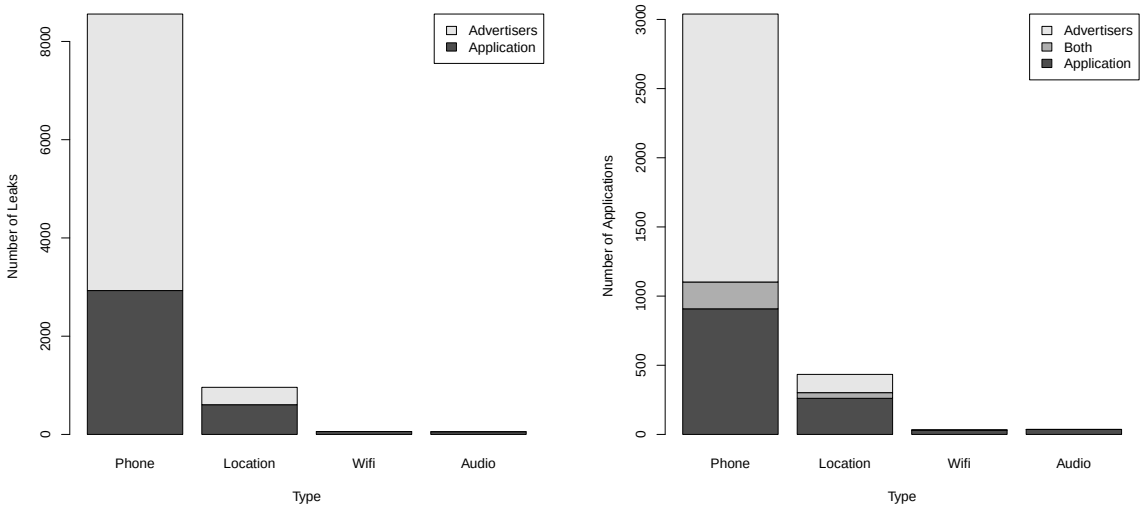
5 Evaluation

We evaluated AndroidLeaks on a body of 23,838 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. To automatically download APKs, we wrote crawlers for both American and Chinese market sites, including SlideMe[16] and GoApk[3]. During crawling, we found that many applications, identified by their SHA1 hashes, are present in multiple markets.

Out of these 23,838 apps we were unable to analyze 4,142 due to invalid bytecodes in the *dex2jar* converted APKs. There were also 1,607 apps which required no permissions. These apps do not have the ability to gain access to sensitive data nor leak information so we exclude them from the analysis described in this section. We found potential privacy leaks in 3,258 of the remaining 18,089 apps.

Using AndroidLeaks on one server-grade computer we were able to analyze all 18,089 apps in under 26 hours- almost 700 APKs per hour. Collectively we processed over 531,000 unique Java classes.

We chose to focus on 4 types of privacy leaks: uniquely identifying phone information, location data, wifi state and recorded audio. This data can be used to uniquely identify a phone and possibly link it to a physical identity. Combined with location and microphone data, a malicious application could record information about the user: who they are, what they do, and where they go.



(a) Unique Leaks

(b) APKs that Leak

Figure 4: Leaks by Type

5.1 Potential Privacy Leaks Found

We found a total of 9,631 leaks in 3,258 Android applications. 2,387 of these are unique leaks, varying by source, sink or code location. 6,156 were leaks found in ad code, which comprises 64% of the total leaks found. In Figure 3 we show the source of leaks of phone and location data, divided into application and ad libraries. Figure 4 shows a breakdown of the leaks we found by leak type. We do not include pie charts for Wifi and record audio leaks because all were found in application code.

5.1.1 Verification

Due to the large number of APKs analyzed and leaks found, it is fundamentally difficult to verify the correctness of all our results due to time constraints. Since APKs are comprised of both ad code and application code, and both may leak, we chose to initially 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 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. Thus we initially focused on verifying the most common unique leaks to determine the veracity of the largest number of leaks. We identify leaks by the 3-tuple: source method, method the source method is called from, and the sink method.

We manually traced 48 leaks in various versions of the Mobclix, adHUBS, Millennial Media, and Mobclick libraries to assess the accuracy of AndroidLeaks's 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. We suspect that the false positives are due to our taint propagating too far and reaching sinks in these other libraries.

The 24 leaks described above are collectively repeated 3057 times and occur in 1606 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." We believe this to be the first findings of sensor data being leaked off the phone.

It is important to note that AndroidLeaks reports potential privacy leaks but cannot automatically verify its results. Manual verification by the application developer or a security researcher is almost always required to determine the veracity of the findings. To ease this process, AndroidLeaks specifies the containing class and method as well as the relevant method call for each leak's source and sink. AndroidLeaks can guide and focus a manual reviewer's time 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 information. 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. Clearly, it's important to determine the types of sensitive data accessed by ad libraries and how it is used.

One solution is to watch an application which uses a given ad library using dynamic analysis, such as TaintDroid. However, one runs into fundamental limitations of dynamic analysis, such as difficulty in achieving high code coverage. Even with maximum possible code coverage using dynamic taint analysis, there is a further problems on Android. Many ad libraries we examined check if the application they were bundled with has a given permission, oftentimes location. Using this information, they could localize ads, potentially increasing ad revenue by increasing click through. However, there is nothing preventing ad libraries for checking if they have access to any number of types of sensitive information and attempting to leaking 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.

Ad libraries tend to be distributed to developers in a precompiled format, so it is not easy for an application developer to determine what information the ad library uses for user analytics. This is important for developers that include ad libraries in highly privileged applications because the developer is ultimately responsible for any information leaked by libraries they choose to include.

Leak Type	Unique Leaks	% of all Leaks	# apps with leak	% apps with leak
Phone	8558	88.9%	2939	16.2%
Location	959	9.96%	434	2.40%
WiFi	59	0.61%	36	0.20%
Record Audio	55	0.57%	32	0.18%

Table 1: Breakdown of Leaks by Type

	Type of Leak					
Ad Library	Phone	Location	Wifi	Microphone	# apps using	% apps using
Mobclix	✓	O	X	X	597	3.3%
Mobclick	✓	O	X	X	436	2.4%
adHUBS	✓	O	X	X	442	2.4%
Millennial Media	✓	O	X	X	162	0.9%

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

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 could be used by Android developers as a tool to help automatically generate the permissions their application needs. Though our mapping is incomplete, the initial results are promising for discovering required permissions are promising. For the following stats, we excluded any developer-defined permissions or permissions internal to Android or Google and not specified on the Android manifest page. On the permissions we focused on for detecting privacy leaks, including INTERNET, READ_PHONE_STATE, and ACCESS_[COARSE—FINE]_LOCATION, we recovered the exact permissions for 14562 out of 16,471, or 88.4%. Over all of the 115 permissions currently defined in the Android documentation, our analysis is able to recover 5468 out of 16,471, or 33.2%. Out of the applications declaring no permissions, our analysis found 82 applications with method calls which require permissions. This is including some permissions we currently have no mappings for and most which we made little effort to improve the mapping for beyond the initial permission mapping creation. One could potentially recover developer defined permissions by examining the permission checks in application code and the filters declared in the application manifest. We leave this problem open for future work.

Likely Developer Permission Errors Android gives developers the flexibility to define per-

missions specific to their application to allow the applications to share functionality with other applications in a mediated fashion. However, as developers must manually specify the permissions their application needs and they are not restricted to the default permissions declared by Android, there is room for developer error. While it's impossible to definitively say that a permission was incorrectly specified without manually reviewing the code, we found a number of permissions that appear to be typographical errors, including "WRITE_EXTERNAL_ STOR**EA**GE," "ACCESS_CO**U**RSE_LOCATION" and "and**oir**d.permission.ACCESS_COARSE_LOCATION". Out of 23,838 there were 551 unique permissions declared. Based solely on the permission names and without manual verification, we estimate at least 125 of these to be developer errors. These findings support the value of our ability to automatically recover an application's required permissions.

Two interesting questions can be raised about these results: 1) are developers over privileging their applications and 2) do developers ever under-privilege their applications. Currently, our incomplete mapping causes us to occasionally miss the requirement of certain permissions, incorrectly leading us to believe an application declares more permissions than it needs. On the other hand, if our mappings are incorrect and we say a method requires a permission when it does not, we may falsely believe an application declares fewer permissions than it needs. These two issues make it difficult to calculate exactly how effective we are at recovering permissions, though these issues do not significantly affect our statistics described above. While we do not know the exact extent of the occurrence of the above two problems, we do have some concrete examples in which developers have not declared permissions their application needs. While these are only a very small percentage of the total applications, they lend credence to the possibility that there may be more instances of both problems and that this functionality would be of use to developers.

5.3 Miscellaneous Findings

Having a large number of Android applications allows us analyze them for other trends such as prevalence and types of ad code or other libraries, frequency of permissions being requested and a number of other statistics. We describe a number of interesting findings in the following sections.

5.3.1 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 which 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.

5.3.2 Native Code

Native code is outside the scope of our analysis, however, it is interesting to see how many applications actually 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 and so the native layer provides no extra functionality. Nevertheless, we found that out of 23,838 applications, 1,457 (6%) of applications have

at least one native code file included in their APK. Of the total 2,652 shared objects in APKs, a majority (1,533, 58%) 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.

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 to users 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 potentially 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 this open to future work.

6.2 Implementation Limitations

Incomplete permission mapping Our mappings 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 out 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.

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, one method can call another, either directly by name or indirectly by type of desired task. Both cases are more complicated to analyze than standard control flow. In the former case, we would need to introspect on the 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.

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. also perform similar analyses with their tool PiOS[13], a static analysis tool for detecting privacy leaks in iOS applications. They ran into a similar inter-procedural problems, where methods were being routed through a dynamic dispatch function in the Objective-C runtime and they had to develop a method to statically follow private data as it propagated through different components of an application. 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 analysis of smartphone applications before this paper, analyzing 1,400 PiOS applications whereas we analyzed over 18,000.

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 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 an interesting approach, it is unlikely that

this patch will be incorporated into stock Android because it may damage Android's economic model. Most free applications are supported by advertising which are driven by user analytics. Therefore, any scheme that disrupts user analytics or advertising will negatively impact developers who monetize their applications through advertising. However, for users capable of flashing their own ROMs, this is potentially a robust way to limit applications.

Enck et. al. in a yet unpublished work, studied 1,100 free Android applications using a commercial tool for static analysis [8]. Because they used a commercial tool but never described its analysis algorithms, it's impossible to compare the merit of analyses directly. From their preliminary results, we can note that Androidleaks is faster and therefore can run on a much larger scale. While just their decompilation took approximately 20 days on 1,100 applications, our conversion and analysis time for 18,000 applications was under 26 hours.

Felt et. al. in a yet unpublished work, investigate permission usage in 940 Android applications using their tool STOWAWAY[9]. In order to determine the API method to permissions mapping, the generated unit tests for each method in the Android API. They then executed these tests and observed whether or a permission was required. This dynamic approach is very precise, however, may be incomplete because dynamic testing may not be run in an environment that requires the permission. Combining their mapping with our statically generated one could produce a very complete and precise mapping.

9 Conclusion

As Android gains even greater market share, its users need a way to determine if personal information is leaked by third-party applications. Whereas iOS incorporates a review and approval process, Android relies on user regulation and 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 over 23,000 Android applications and detected over 9000 potential privacy leaks in over 3,200 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. http://www.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-48.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.