SAND2011-4758C

AndroidLeaks: Detecting Privacy Leaks In Android Applications

Clint Gibler
cdgibler@ucdavis.edu

Jonathan Crussell
jcrussell@ucdavis.edu

Jeremy Erickson
jericks@ucdavis.edu

Hao Chen
hchen@cs.ucdavis.edu

Categories and Subject Descriptors
security [**privacy**]: Android

General Terms
static analysis, Android

Keywords
taint analysis, static analysis, Android, privacy

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]. As such, it is important to ensure that smartphones running the Android operating system maintain an acceptable level of security for a user's private information.

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[10], 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 your 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 personal 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. Using WALA[5], a program analysis framework for Java source and byte code, we create a call-graph for the application code and then perform a reachability analysis to determine if sensitive information may be sent over the network. In cases where it can, we use dataflow analysis to track tainted data to see if it is ever leaked.

Other projects, such as TISSA[17], 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 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. While several other tools exist to find privacy violations in Android applications[7, 8], 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, precisely recovering the subset of permissions we focused on 88.4% of the time and precisely recovering the permissions 33.2% of the time across all standard Android 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 independent collection 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 several popular ad libraries in terms of the permissions they use and types of data they leak. We also analyzed their leaks and then manually verified them so that we can 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 [13].

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 [12, 16]. 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

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

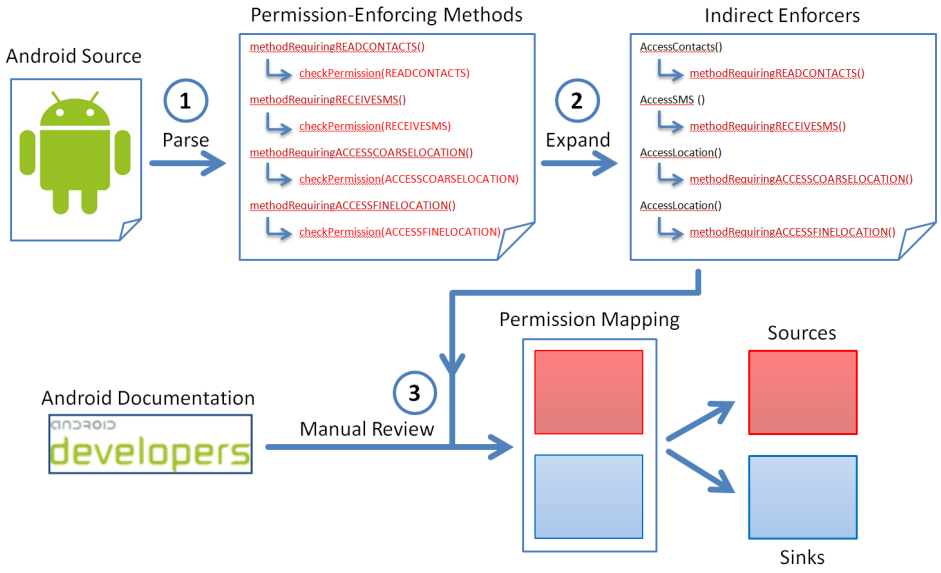


Figure 1: Creating a Mapping between API Methods and Permissions.

permissions. The user will be able to install the application only if she grants the application all the permissions. Currently, there is 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

Our work focuses on the threat of 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.

Our goal is to determine if applications are using the permissions they are granted to leak sensitive data to a third-party. 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 evaluate whether or not a leak is required for application functionality.

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.

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

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 directly enforce a permis-

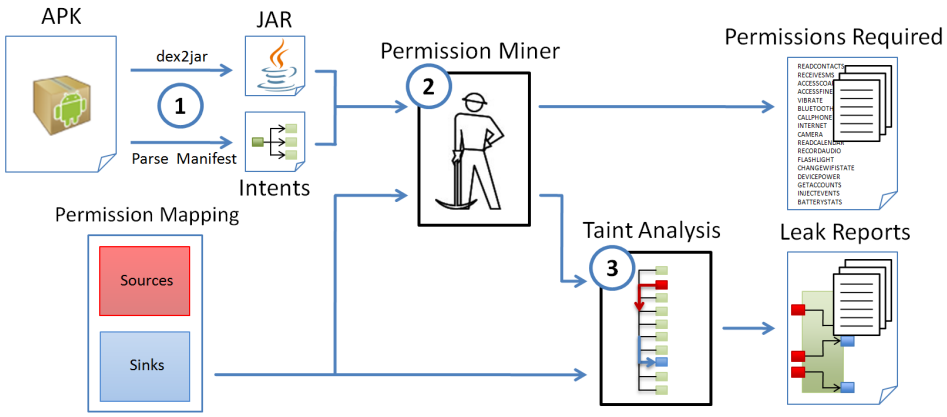


Figure 2: AndroidLeaks Analysis Process.
1. Preprocessing. 2. Recursive call stack generation to determine where permissions are required. 3. Dataflow analysis between sources and sinks.

sion, such as *PermissionController.checkPermission(String permission, int pid, int uid)*. By parsing the Android source code for occurrences of known permission identifiers, we were able to find the methods in which these helper functions were used.

However, few of these methods are part of the Android API accessable to the developer. Thus, we needed to propagate the method call to permission mapping up to the Android SDK method calls that developers use. For every method in every class, we recursively determined the methods called by each method in the framework, building a call stack of the Android source. If our analysis encountered a method in our permission to API method mapping, we labeled the method as requiring that permission as well as each parent method in the call stack we had been building.

To supplement our programmatic analysis, we manually reviewed the Android documentation to add mappings we may have missed. Currently we have mappings between over 2000 methods and the permissions they require. 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.

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*[13]. 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 callgraph 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, the ACCESS_FINE_LOCATION permission supercedes the ACCESS_COARSE_LOCATION permission. Specifically, 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 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 to third parties 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 Android applications. 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 [15], 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.

In particular, our third taint propagation rule was a large source of false positives, especially when applied to Activities. As mentioned before, Activities are responsible for a single UI screen. This screen is comprised of several views which are UI foundations, such as text views and buttons. Some advertising libraries can be bound directly to a view by the Activity for ease of use. Therefore, if the Activity object ever becomes tainted, we will report a leak because some data from the Activity flows to the advertisers and therefore to a sink.

5. EVALUATION

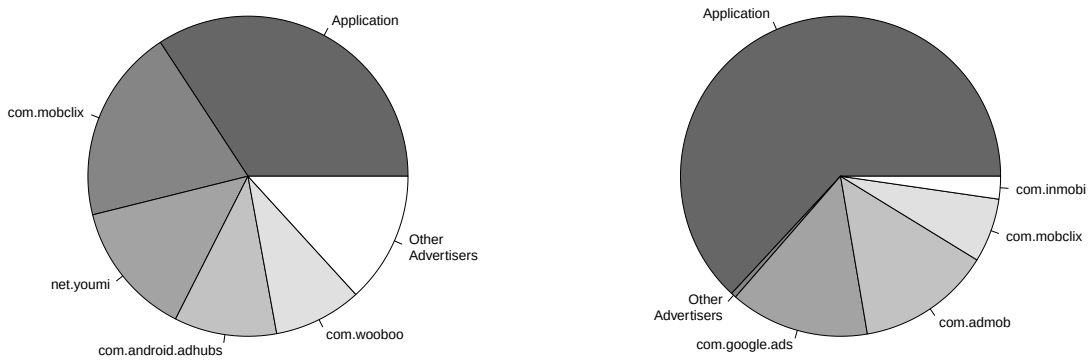
We evaluated AndroidLeaks on a body of 23,838 unique Android applications, including nine applications that included the Geinimi library, which is known to be malware. The official Android Market[9] has many free applications but Google has created mechanisms to discourage automated crawling. Fortunately, the application distribution model of Android applications works in our favor — there are many third-party Android market sites. We wrote crawling scripts for both American and Chinese market sites, including SlideMe[14] and GoApk[3]. We found that many applications are present in multiple markets. We identified identical applications across markets by comparing their SHA1 hashes.

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. Of the remaining 18,089 apps, we found potential privacy leaks in 3,258 of them, approximately 18%.

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.

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,



(a) Phone Identification

(b) Location

Figure 3: Leaks by Source Location

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. 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 almost a third of the total discovered leaks are true positives and about half of the total reported leaky APKs have confirmed leaks.

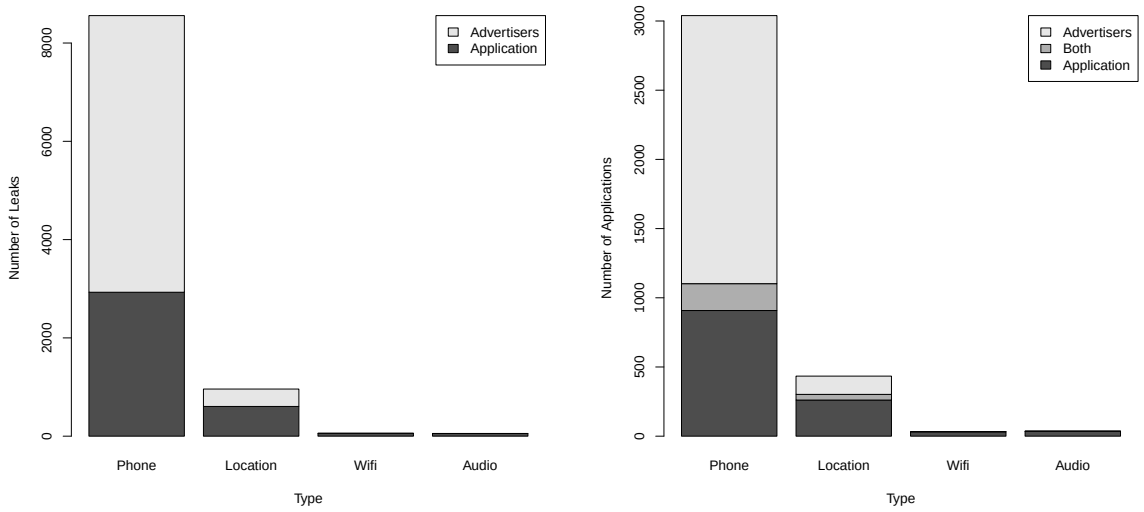
We also verified a small random set of applications containing each leak type in application code to confirm AndroidLeaks is successful at finding leaks in application code as well. Interestingly, several of the microphone leaks we verified turned out to be in IP camera applications, such as "SuperCam" or "IP Cam Viewer Lite." We believe that the leak paths we reviewed were for legitimate application functionality, but it's difficult to determine intention with certainty.

Our manual verification confirmed if leaks we found were real but we did not attempt to determine something equally interesting: if the leaks were a part of described application functionality or if they were potentially malicious. Differentiating the two was not a focus of our tool and it is difficult to determine this even manually without becoming familiar with the structure of the application and the purpose of its components. Clearly the latter is impractical to do on a representative body of applications.

5.1.2 Ad Libraries

During our manual review of ad libraries, we found numerous instances of leaks that our analysis did not find. 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 informa-



(a) Unique Leaks

(b) APKs that Leak

Figure 4: Leaks by Type

tion. 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 problem specific to 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 throughs. 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 if an ad library is malicious through manual review. 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. There is no current mechanism in Android allowing one to restrict the capability 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 still far from precise or complete, the initial results 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 permissions specific to their application to allow the applications to share functionality with other applications in a mediated fashion. However,

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

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 arise in response to the permission generation problem: first, are there cases in which developers declare more permissions than they need? Secondly, do developers ever declare fewer permissions than they need? 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. We may overestimate this value because we consider all code to be potentially callable in order to not miss cases where our lack of Android specific control flow would make us miss leaks. 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

There are currently several limitations of our design and implementation, both will be discussed below and possible

improvements will be discussed in Section 7

6.1 Implementation Limitations

Incomplete permission mapping Our mappings between API methods and permissions is not complete. This will lead to our analysis underreporting the permissions required by some applications since we will not recognize certain API calls as requiring permissions. This also leads to us having an incomplete set of sources for our privacy leak analysis which may cause us to miss privacy leaks. This issue may or may not be easily solved, and will be discussed in Section 7.

Android-specific control and data flows We do not 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. These Android-specific flows have been investigated in SCanDroid [8] and their approaches could be added to our analysis in order to detect more complex leaks, possibly involving multiple applications colluding.

6.2 Design Limitations

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 even begin analyzing an application. We were unable to analyze approximately 17 % of our applications due to problems with dependencies.

Native code Finally, another limitation of our design is that we do not consider privacy leaks in native code. As discussed in Section 5.3.2, a portion of applications use native code. Because our analysis is restricted to Java bytecode, our taint analysis will not be able to track data flows into native code. In the worst case, an application could be almost entirely written in native code, and only use Java to access the Android APIs not available in native libraries. Our current analysis would be able to report very little about such an application.

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 Though we have made every effort to be accurate and precise in our mappings of API methods to required permissions, we are missing a number of methods that require permissions and we have likely included at least several that do not require permissions. These are sources of false negatives and false positives, respectively. There are certain fundamental aspects of static analysis that can cause implementations to either lose soundness or completeness. While we cannot avoid fundamental limitations of static analysis, we assuredly have control over the accuracy and precision of our mapping. We plan to improve our mapping so that we can minimize these unnecessary imprecisions.

8. RELATED WORK

Chaudhuri et. al. present a methodology for static analysis of Android applications to help identify privacy violations in Android with SCanDroid[8]. They used WALA to analyze the source code of applications, rather than the 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[11], 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 23000. .

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. TaintDroid is limited in the number of applications that can be tested because they must all be tested through execution. This approach requires that a user manually click through the application windows, trying to trigger data leaks. Our framework, which uses static analysis, can analyze many more applications.

Zho et. al. presented a patch to the Android operating system that would allow users to selectively grant permissions to applications [17]. 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. However, for users capable of flashing their own ROMs, this is potentially a very robust way to limit applications.

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 re-

stricted 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.

10. 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] 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.

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

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

[11] 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.

[12] 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.

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

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

[15] 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.

[16] 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.

[17] 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.