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

Clint Gibler[1], Jonathan Crussell[1,2], Jeremy Erickson[1,2], and Hao Chen[1]

[1] University of California, Davis
{cdgibler, jcrussell, jericks}@ucdavis.edu, hchen@cs.ucdavis.edu
[2] Sandia National Labs*, Livermore, CA
{jcrusse, jericks}@sandia.gov

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

Google's Android operating system provides a permissions-based security model that restricts an application's access to the user's private data. Each application statically declares the sensitive data and functionality that it requires in a manifest, which is presented to the user upon installation. However, it is not clear to the user how sensitive data is used once the application is installed. To combat this problem, we present AndroidLeaks, a static analysis framework for automatically finding potential leaks of sensitive information in Android applications on a massive scale. AndroidLeaks drastically reduces the number of applications and the number of traces that a security auditor has to verify manually.

We evaluate the efficacy of AndroidLeaks on 24,350 Android applications from several Android markets. AndroidLeaks found 57,299 potential privacy leaks in 7,414 Android applications, out of which we have manually verified that 2,342 applications leak private data including phone information, GPS location, WiFi data, and audio recorded with the microphone. AndroidLeaks examined these applications in 30 hours, which indicates that it is capable of scaling to the increasingly large set of available applications.

1 Introduction

As smartphones have become more popular, the focus of mobile computing has shifted from laptops to phones and tablets. There are several competing mobile platforms. As of this writing, Android has the highest market share of any

* Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

smartphone operating system in the U.S. [8]. Android provides the core smartphone experience, but much of a user's productivity depends on third-party applications. To this end, Android has numerous marketplaces where users can download third-party applications. In contrast to the market policy for iOS, in which every application is reviewed before it can be published [15], most Android markets allow developers to post their applications 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 and reported malicious applications removed.

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. 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 these permissions, 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's functionality requires sending location data to the maps server, which is expected and acceptable given the purpose of the application. However, if the application is ad-supported it may also leak location data to advertisers for targeted ads, which may compromise a user's privacy. Given the only information currently presented to users is a list of required permissions, a user will not be able to tell how the maps application is handling her location information.

To address this issue, we present AndroidLeaks, a static analysis framework designed to identify potential leaks of personal information in Android applications on a large scale. Leveraging WALA [7], a program analysis framework for Java source and byte code, we create a call graph of an application's code and then perform a reachability analysis to determine if sensitive information may be sent over the network. If there is a potential path, we use dataflow analysis to determine if private data reaches a network sink.

Our contributions in this paper are as follows:

- We have created a set of mappings between Android API methods and the permissions they require to execute using static techniques. We use a subset of this mapping as the sources and sinks of private data for our dataflow analysis.
- We present AndroidLeaks, a static analysis framework for finding potential leaks of private information in Android applications. We evaluated An-

droidLeaks on 24,350 Android applications, finding potential privacy leaks involving uniquely identifying phone information, location data, WiFi data, and audio recorded with the microphone. AndroidLeaks identifies APKs and provides a set of leaks most likely to be of interest to a security researcher.
- We designed and implemented taint-aware slicing and an approach for identifying taint sources in callbacks, which is used extensively in Android applications.
- We compare the prevalence of several popular ad libraries and the private data they leak.

2 Background

Android applications are primarily written in Java. Unlike standard Java applications, after being compiled into Java bytecode Android applications are converted into the Dalvik Executable (DEX) format. This conversion occurs because Android applications run in the Dalvik [6] virtual machine, rather than the Java virtual machine. We use *ded* [11] and *dex2jar* [17] to convert applications back into Java source code or byte code, respectively.

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

The official Android Market is primarily user regulated. The ratings of applications in the market are determined by the positive and negative votes of users. Higher ranked applications are shown first in the market and therefore are more likely to be discovered. Users can also share their experiences with an application by submitting a review. This can alert other users to avoid poorly behaving applications. Google is able to remove any application not only from the market, but also from users' phones directly, and has done so when users reported malicious applications [16, 20]. However, recent research [10] 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[3] to application-managed data. In order to communicate and coordinate between components, Android provides a message routing system based on URIs. The sent messages are called Intents. Intents can tell the Android framework to start a new Service, switch to a different Activity, or to pass data to another component.

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

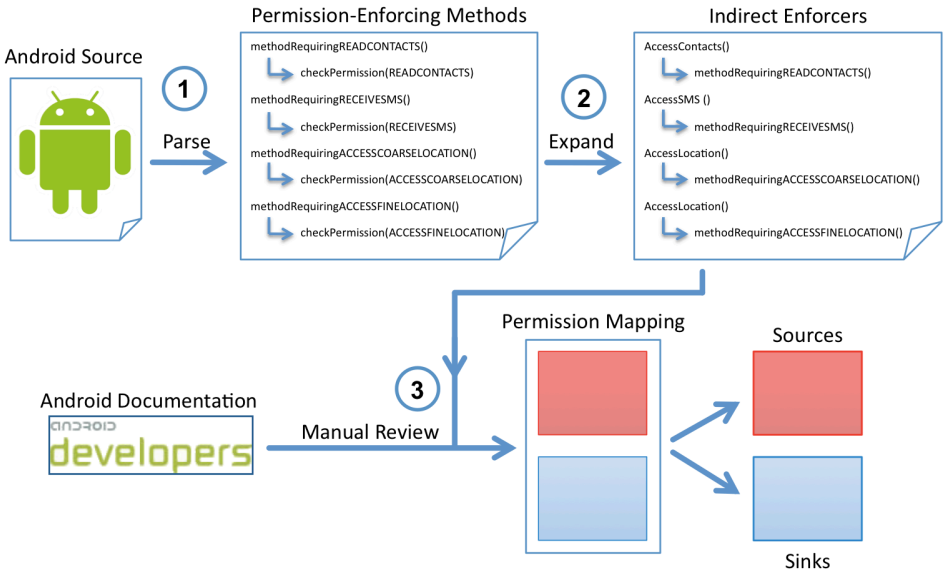


Fig. 1: Creating a Mapping between API Methods and Permissions.

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

3 Threat Model

In this work we consider a *privacy leak* to be any transfer of personal or phone-identifying information off of the phone. We do not attempt to distinguish per-

sonal 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. Identifying if personal data is used for expected functionality requires understanding the purpose of the application as well as the intention of the developer during its creation, neither of which we attempt to do. Thus we classify transfer of personal information off of the phone as a privacy leak regardless of its use, e.g., malware authors may maliciously leak private data, ad libraries may leak it for more targeted ads, and applications may use it for their functionality. We focus on tracking private information flow in real applications at a large scale, but leave determining the intent of private information leaks 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 which runs applications. For example, a Webkit[4] 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 Linux kernel and libraries, the Android framework, and the Dalvik VM.

We do not attempt to track private data specific to an application, such as saved preferences or files, since determining which application-specific data is private requires knowledge of the application's purpose and therefore is difficult to automate. We also do not attempt to find leaks enabled by the collaboration of applications. To find such leaks, we would need to extend AndroidLeaks to analyze potential interactions between applications, which we leave for future work.

Currently AndroidLeaks does not analyze native code. We do not believe this significantly affects our results as only 7% of our Android applications include native code. Even if an application is written in native code to defeat Java-based analyses such as AndroidLeaks, it cannot hide its access of private data because it may read private data only through Android's Java APIs. AndroidLeaks could be extended so that, when an application reads private data and then passes it to native code, AndroidLeaks would pass the analysis on to existing binary analysis tools, such as BitBlaze [3].

4 Methodology

In this section we discuss the architecture and implementation of AndroidLeaks. First, we create a *permission mapping* — a mapping between Android API calls and permissions they require to execute — to be used in all application analyses. We use a subset of this mapping for our dataflow *sources* and *sinks*. A *source* is a method that accesses personal data; for example, a phone number, unique device ID, or the phone's GPS location. A *sink* is a method that can transmit local data to an external entity; for instance, submitting a HTTP request. Next, for

[4] Webkit is a rendering engine used by Android's browser.

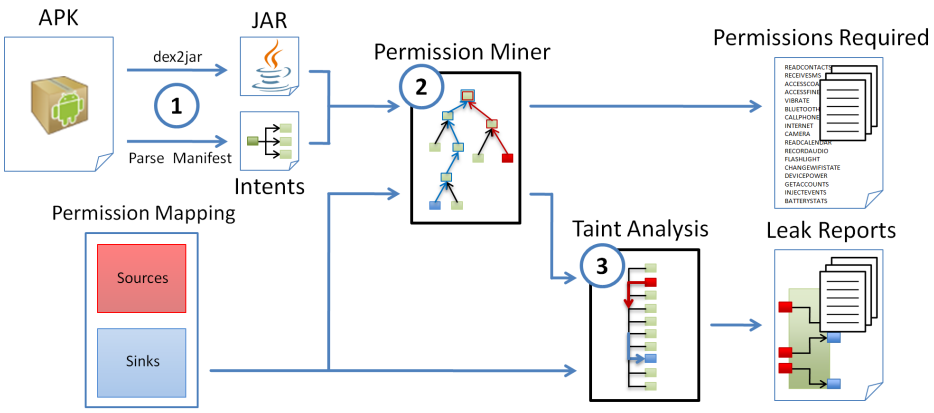


Fig. 2: AndroidLeaks Analysis Process. 1. Preprocessing. 2. Recursive call stack generation to determine where permissions are required. 3. Dataflow analysis between sources and sinks.

each application, AndroidLeaks generates a call graph to determine the call sites which invoke source or sink methods. Applications without at least one source and sink are not analyzed, as they cannot leak private data. For applications that have the potential to leak, we perform static taint analysis to determine if data from a source method reaches a sink.

4.1 Permission Mapping

To determine if an application is leaking sensitive data, first one must define what should be considered sensitive. Intuition and common sense may give a good starting point; however, in Android we can do much better since access to restricted resources is protected by permissions. Of these restricted resources, some control access to sensitive data, such as precise geographic location. It is likely that API calls that require sensitive permissions 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. It would be useful for developers because it would help them better understand the permissions required by their desired functionality. Unfortunately, the Android documentation is incomplete, and only a partial mapping is provided. To address this issue, we attempt to automatically build this mapping by directly analyzing the Android framework source code. Figure 1 visualizes our process.

Intuitively, for a permission to protect restricted functionality, there must be points in the code where the permission is checked. In manual analysis of the Android source, we found a number of helper functions that enforce a permission, such as *Context.enforcePermission(String, int, int)*, where the first parameter is the name of the permission. For every method in every class of the Android

framework, we recursively determined the methods called by each method in the framework, building a call stack, a process we call *mining*. Our miner will use all possible targets of virtual methods, erring on the side of completeness, rather than precision. If our mining encounters one of these enforcement methods, we inspect the value of the first parameter in order to determine the name of the permission being enforced. We then propagate the permission requirement to all the methods in the current call stack. After the permission mining is complete, we have a mapping between methods and the permissions they require. A subset of the methods in this mapping are API methods which are directly available to developers through the SDK.

Though this process gave us many mappings, it does not find permission checks that are implemented outside the Android framework and can not propagate permission requirements along edges connected by Intents or by IPC to a system process. To supplement our programmatic analysis, we manually reviewed the Android documentation to add mappings we may have missed. While this may seem significant, we note 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. Additionally, at some points in the Android framework, it may check, but not enforce a permission using a method such as *Context.checkPermission(String, int, int)*. For each of these points in the code, we determined how the check was used and what method actually requires that permission and add it to our permission mapping before the mining process. Currently we have mappings between over 2000 methods and the permissions they require. To check the completeness of our mapping, we plan to collaborate with the group that worked on [12], which has also created a permission mapping but with dynamic testing.

4.2 Android Leaks

In this section we describe AndroidLeaks' analysis process. See Fig. 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 *ded* [11] or *dex2jar* [17]. AndroidLeaks can also use any other tool that converts DEX to a JAR or to Java source.

Using WALA, AndroidLeaks then builds a call graph of the application code and its included libraries. It iterates through the application classes and determines the application methods that call *source* and *sink* API methods. It also keeps track of which other application methods can call these application methods that require permissions, as reviewing the call stacks can give insight into the flow of the application's use of permissions. If the application contains a combination of permissions that could leak private data, such as READ_PHONE_STATE and INTERNET, it then performs dataflow analysis to determine if information from a source of private data may reach a network sink.

Taint Problem Setup The two main components of taint problems are determining the sources and sinks.

Sources We have selected all the API methods requiring permissions for location, network state, phone state, and audio recording as sources, as discussed in Sect. 4.1. Android has two categories of location data: coarse and fine. Coarse location data uses triangulation from the cellular network towers and nearby wireless networks to approximate a device's location, whereas fine location data uses the GPS module on the device itself. We do not differentiate between coarse and fine location data as we believe any leakage of location information to be important.

Sinks We have selected methods that require access to the Internet as sinks. We discovered that the Internet permission is enforced by the Android sandbox, which will cause any open socket command to fail if the INTERNET permission has not been granted. As discussed in Sect. 4.1, we manually reviewed the standard APIs available to Android applications to ensure our mapping contained every method that allows an application to send network data.

Taint Analysis First, we use WALA to construct a context-sensitive System Dependence Graph (SDG). Since context-sensitive pointer analysis is resource intensive, we chose to use a context-insensitive overlay to show heap dependencies in the SDG. The SDG is a graph that describes the inter- and intraprocedural control and data dependencies of an application. Using the SDG, for each source method, we compute forward slices from our set of tainted data, initially populated by the return value of the source method. We use the return value because all the sources that we have identified return sensitive data through the return values only (and not through other means, such as side-effects on the parameters). On each iteration, we obtain a new slice of tainted data to which we apply supplemental taint-forwarding procedures. We then analyze the slice to determine if any parameters to sink methods are tainted, i.e., if they are data dependent on the source method. If so, we report a potential leak of private data.

WALA's built-in SDG and forward slicing algorithms are insufficient for analyzing Android applications, because they fail to handle callbacks, which are used extensively in Android applications, or do taint-aware slicing.

Handling Callbacks Private data may enter Android applications via API methods identified as sources in Sect. 4.2. However, they may also enter applications via callback parameters, which are used extensively in Android. For example, an application may access location information either by asking the LocationManager for the last known location or by registering with the LocationManager as a listener. For the latter, the LocationManager provides regular updates of the current location to the registered listener. For API methods labeled as sources, we can taint the return values of these methods; however, this approach does

not work for callbacks since neither the return value of the callback nor the return value of the registration is tainted. Therefore, we automatically identified calls to the register listener method while mining the application code and then inspected the parameters to determine the type of the listener. We then tainted the parameters of the callback method for the listener's class. This approach allows us to compute forward slices for both types of access in the same way.

Taint-Aware Slicing Rather than modify WALA internally as done in [19] to achieve taint-aware slicing, we decided to analyze the computed slices and compute new statements from which to slice. We implemented the following logic to compute these new statements:

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

By applying these propagation rules to the slice computed for the source method, we create a set of statements that are tainted but would not be included in the original slice. This is because the original slice only shows statements that are data dependent, which is only part of how taint propagates. We then compute forward slices for each of these new 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.

Preventing over-tainting without missing taint propagation is a difficult problem in static analysis, especially when complex objects handle both tainted and untainted data. Since we do not wish to miss any taint propagation, we conservatively track all potential taint propagation, which may result in false positives. We note that [19] also has high false positives in certain cases.

5 Evaluation

We evaluated AndroidLeaks on 25,976 unique free Android applications obtained from thirteen Android markets, including the official Android Market [14] and third-party American and Chinese markets.[5] We exclude multiple versions of the same application and duplicate copies of the same application on multiple markets.

1,626 applications require no permissions. Since these applications cannot access private data nor leak it, we exclude them from the analysis. We found potential privacy leaks in 7,414 of the remaining 24,350 applications.

Running AndroidLeaks on one server-grade computer we were able to analyze all 24,350 applications in 30 hours- over 800 APKs per hour. Collectively we processed over 531,249 unique Java classes.

We chose to focus on 4 types of privacy leaks: uniquely identifying phone information, location data, WiFi state and recorded audio. Examples of uniquely

[5] Including SlideMe [18] and GoApk [4].

Table 1: Breakdown of Leaks by Type

Leak Type	# Leaks	% of all Leaks	# apps with leak	% apps with leak
Phone	53,281	92.99%	6,912	28.39%
Location	3,405	5.94%	969	3.98%
WiFi	266	0.46%	79	0.32%
Record Audio	347	0.61%	115	0.47%

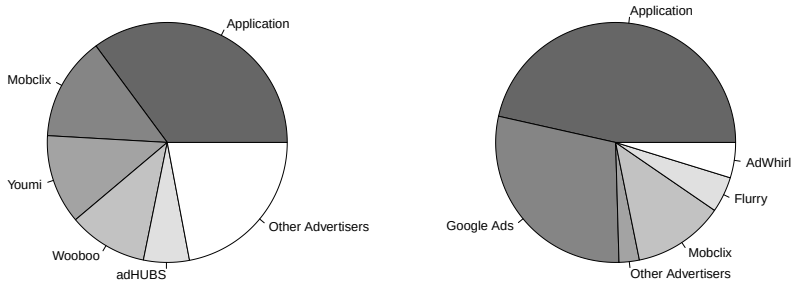
identifying phone information include the unique device ID (IMEI for GSM phones, MEID or ESN for CDMA phones) and the subscriber ID (IMSI for GSM phones). For location data, AndroidLeaks tracks accesses to both "coarse" and "fine" GPS data. WiFi state information includes the SSID and BSSID of the current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

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

5.1 Potential Privacy Leaks Found

We found a total of 57,299 leaks in 7,414 Android applications. 7,870 of these are unique leaks, varying by source, sink or code location (Table 1). 36,388 were leaks found in ad code, which comprises 63.51% of the total leaks found. In Fig. 3 we show the source of leaks of phone and location data, divided into leaks found in application code and ad libraries. We do not include pie charts for WiFi and record audio leaks because all of these leaks were found in application code. Ad libraries were responsible for 65% of the total phone data-related leaks with the top four ad libraries accounting for 43%. Application code contained 46% of the location-related privacy leaks and the top four ad libraries were responsible for 51%. Figure 4a shows a breakdown of the leaks found by the type of leak and its source. Figure 4b displays the number of applications we found containing each type of leak, organized by the source of the leak. We found that in most cases where phone identifying information is leaked, the advertising library is solely responsible.

Verification Due to the large number of APKs analyzed and leaks found, it is difficult to manually verify all the leaks. Therefore, we prioritize the task by initially focusing on verifying leaks in ad code. By verifying one leak in a given ad library we can extend that result to identical leaks in other applications containing the same version of the same ad library. We determine leaks to be



(a) Sources of leaks of phone id

(b) Sources of leaks of location information

Fig. 3: Source of leaks

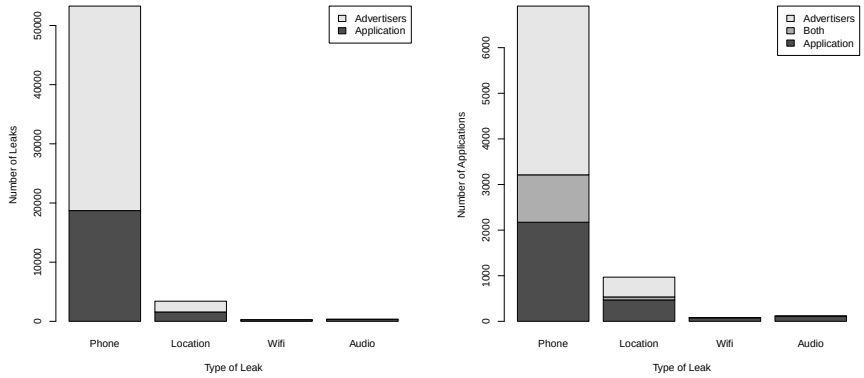
identical if they share the same source and sink method as well as the class and method where each is called.

We manually verified 60 leaks, most of which occurred in the ad libraries shown in Fig. 3. Of these, we found 39 to be true positives, yielding a false positive rate of 35%. The false positives tended to occur most commonly in applications that contained ad libraries in addition to the one containing the leak being verified. As multiple ad libraries may populate UI components on the same screen, our analysis may conservatively say that it is possible for sensitive data accessed by one ad library to propagate to its containing Activity or other ad libraries that share the same Activity. The 39 leaks we verified are repeated 5,007 times and occur in 2,342 unique applications. Therefore, at least 32% of the leaky APKs AndroidLeaks found have confirmed leaks.

Additionally, we verified a random set of 15 applications collectively containing several leaks of each type in application code. Several of the microphone leaks we verified turned out to be in IP camera applications, such as "SuperCam" or "IP Cam Viewer Lite." Figure 5 and Table 3 show the total number of verified leaks and leaky applications.

After AndroidLeaks reports potential privacy leaks, a security auditor can manually verify these leaks. To help with the manual verification, AndroidLeaks specifies the containing class and method as well as each leak's source and sink.

Ad Libraries Nearly every ad library we looked at leaked phone data and, if available, 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.



(a) Number of unique leaks broken down by their sources

(b) Number of applications that leak in ad code, app code, or both

Fig. 4: Number of unique leaks and leaky applications

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

One solution is to watch an application that uses a given ad library using dynamic analysis, such as TaintDroid. However, one runs into limitations of dynamic analysis, such as difficulty in achieving high code coverage. Manually driving applications through all code paths is infeasible at the rate new Android applications are being published, between 7,500 and 22,500 per month according to [5]. But even with maximum possible code coverage using dynamic taint analysis, there are further challenges on Android. Many ad libraries we examined check if the application they were bundled with has a given permission, oftentimes the ability to access location data. Using this information, they could localize ads, potentially increasing ad revenue by improving click through rates. However, there is nothing preventing ad libraries from checking if they have access to any number of types of sensitive information and attempting to leak them only if they are able. A dynamic analysis approach could watch many applications with a malicious advertising library and never see this functionality if none of the applications declared the relevant permissions. Using our static analysis approach we do not have this limitation and would be able to find these leaks regardless of the permissions required by the application being analyzed.

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

Table 2: Verified number of unique leaks and leaky applications

Leak Type	# verified leaks	# apps with verified leak
Phone	3731 (84.91%)	2083 (8.55%)
Location	646 (14.70%)	323 (1.33%)
WiFi	0 (0%)	0 (0%)
Record Audio	17 (0.39%)	9 (0.04%)

(a) Verified number of unique leaks (b) Verified number of leaky applications

Fig. 5: Verified number of unique leaks and leaky applications

tionally, 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 sandboxing 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.

Table 2 and Fig. 5 shows the total verified number of unique leaks and number of leaky applications.

Table 3 shows the number of unique leaks of each data type in the 15 applications that we manually verified. Of these data types, device ID, subscriber ID, line one number, and SIM serial number all uniquely identify a phone.

After AndroidLeaks reports potential privacy leaks, a security auditor can manually verify these leaks. To help with the manual verification, AndroidLeaks specifies the containing class and method as well as each leak's source and sink. AndroidLeaks drastically reduces the number of applications and the number of traces that a security auditor needs to verify manually.

Table 3: Number of leaks by data type in 15 manually verified applications

Leak Type	# Verified leaks
Device ID	9
Line 1 Number	3
Subscriber ID	2
SIM Serial Number	2
Other Phone Data	10
Location Data	9
Recorded Audio	4

5.2 Miscellaneous Findings

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

Native Code Native code is outside the scope of our analysis, however, it is interesting to see how many applications use native code. The use of native code is discouraged by Android as it increases complexity and may not always result in performance improvements. Additionally, all Android APIs are accessible to developers at the Java layer, so the native layer provides no extra functionality. We found that 1,988 out of 25,976 applications (7%) have at least one native code file included in their APK. Of the total 3,902 shared objects in APKs, a majority (2,014, 52%) of them were not stripped. This is interesting because stripping has long been used to reduce the size of shared libraries and to make them more difficult to reverse engineer, however, a majority of the applications we downloaded contained unstripped shared objects. This may be a result of developers using C/C++ who aren't familiar with creating libraries.

6 Limitations

Approach Limitations There are several inherent limitations to static analysis. Tradeoffs are often made between speed, precision, and false positives. AndroidLeaks errs on the side of false positives rather than false negatives, as we intend AndroidLeaks to provide potential leaks to security auditors.

While a dynamic approach would have high precision due to the fact that privacy leaks are directly observed at run-time, achieving high path coverage is challenging. Moreover, dynamic analysis tools [10] tend to be manually driven,

which does not scale to the massive number of Android applications. Combining AndroidLeaks with a dynamic approach would have great potential, as AndroidLeaks can quickly analyze a larger number of applications and then feed potential leaky applications to further dynamic analysis. We leave combining AndroidLeaks with a dynamic analysis approach for future work.

Implementation Limitations 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.

7 Related Work

Chaudhuri et al. present a methodology for static analysis of Android applications to help identify privacy violations in Android with SCanDroid [13]. They used WALA to analyze the source code of applications, rather than Java byte code as we do. While their paper described mechanisms to handle Android specific control flow paths such as Intents which our work does not yet handle, their analysis was not tested on real Android applications.

Egele et al. perform similar analyses with their tool PiOS [9], a static analysis tool for detecting privacy leaks in iOS applications. AndroidLeaks and PiOS both found privacy leaks related to device ID, location and phone number. PiOS additionally considered the address book, browser history and photos while we consider several other types of phone data, WiFi data and audio recorded with the microphone. PiOS ignored leaks in ad libraries, claiming that they always leak, while one of the focuses of our work is giving developers insights into the behavior of ad libraries.

In comparison to AndroidLeaks's static analysis approach, TaintDroid [10] 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 having a higher false positive rate. One benefit of AndroidLeaks over the implementation of TaintDroid is that AndroidLeaks is entirely automated, while TaintDroid requires manual user interaction to trigger data leaks. We believe that AndroidLeaks and TaintDroid are in fact complementary approaches, AndroidLeaks can be used to quickly eliminate applications from consideration for dynamic testing while flagging areas to test on applications that are not eliminated.

Zho et al. presented a patch to the Android operating system that would allow users to selectively grant permissions to applications [21]. Their patch gives users the ability to revoke access to, falsify, or anonymize private data.

While this is an effective way to limit permissions granted to applications, it requires flashing the phone's ROM, which voids most phone warranties and is too technical for many users.

Enck et al. [11] created *ded*, a tool that decompiles DEX to Java source code. They used *ded* to convert 1,100 free Android applications to Java source code that they then analyzed with a commercial static analysis tool. Because they used a commercial tool but never described its analysis algorithms, it is difficult to compare the merit of our analyses directly. From their preliminary results, we can note that Androidleaks is faster and therefore can run on a much larger scale. While just *ded*'s decompilation took approximately 20 days on 1,100 applications, our conversion and analysis time for 24,000 applications was approximately 30 hours. Their analysis time was not specified.

Felt et al. investigated permission usage in 940 Android applications using their tool STOWAWAY [12]. In order to determine the API method to permissions mapping, they generated unit tests for each method in the Android API and observed if the execution caused a permission check. This dynamic approach is very precise, however, it may be incomplete if the automated test construction failed to call API methods with arguments that cause the method to perform a permission check. Selectively combining their mapping with our statically generated one could produce a very complete and precise mapping.

8 Conclusion

Android users need a way to determine if applications are leaking their personal information. To this end we present AndroidLeaks, a static analysis tool for finding potential privacy leaks in Android applications. In order to make AndroidLeaks, we created a mapping between API calls and the permissions they require. AndroidLeaks is scalable to the current rate of new applications being submitted to markets, capable of analyzing 24,350 in 30 hours. During analysis, AndroidLeaks found 57,299 potential privacy leaks in over 7,400 applications, out of which we have manually verified that 2,342 applications leak private data. AndroidLeaks drastically reduces the number of applications and the number of traces that a security auditor has to verify manually.

9 Acknowledgments

The authors would like to thank Ben Sanders and Justin Horton for helping us obtain Android applications and our anonymous reviewers for their input. This material is based in part upon work supported by the National Science Foundation under Grant Numbers 0644450 and 1018964. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

1. Android developer reference. Accessed March 30, 2012. http://d.android.com/.
2. Android security and permissions. Accessed March 30, 2012. http://d.android.com/guide/topics/security/security.html.
3. Bitblaze. http://bitblaze.cs.berkeley.edu/.
4. Go Apk. Go apk market. Accessed March, 2011. http://market.goapk.com.
5. AppBrain. Number of available android applications. Accessed August 15, 2011. http://www.appbrain.com/stats/number-of-android-apps.
6. Dan Bornstein. Dalvik vm internals, 2008. Accessed March 18, 2011. http://goo.gl/knN9n.
7. IBM T.J. Watson Research Center. T.j. watson libraries for analysis (wala), March 2011. Accessed March 30th, 2012.
8. 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.
9. M. Egele, C. Kruegel, E. Kirda, and G. Vigna. Pios: Detecting privacy leaks in ios applications. In *Proceedings of the Network and Distributed System Security Symposium*, 2011.
10. W. Enck, P. Gilbert, B.G. Chun, L.P. Cox, J. Jung, P. McDaniel, and A.N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6. USENIX Association, 2010.
11. W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of android application security. In *Proc. of the 20th USENIX Security Symposium*, 2011.
12. A.P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.
13. A.P. Fuchs, A. Chaudhuri, and J.S. Foster. Scandroid: Automated security certification of android applications. *Manuscript, Univ. of Maryland, http://www.cs.umd.edu/ avik/projects/scandroidascaa*, 2009.
14. Google. Google play. Accessed March, 2011. http://market.android.com.
15. Apple Inc. App store review guidelines. Accessed March 30th, 2012. http://developer.apple.com/appstore/guidelines.html.
16. 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.
17. pxb1988. dex2jar: A tool for converting android's .dex format to java's .class format. Accessed March 30th, 2012. https://code.google.com/p/dex2jar/.
18. SlideMe. Slideme: Android community and application marketplace. Accessed March 30th, 2012. http://slideme.org/.
19. O. Tripp, M. Pistoia, S.J. Fink, M. Sridharan, and O. Weisman. Taj: effective taint analysis of web applications. In *ACM Sigplan Notices*, volume 44, pages 87–97. ACM, 2009.
20. 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.
21. Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming information-stealing smartphone applications (on android). *Trust and Trustworthy Computing*, pages 93–107, 2011.