Unsafe Exposure Analysis of Mobile In-App Advertisements

Michael Grace, Wu Zhou, and Xuxian Jiang
Department of Computer Science,
North Carolina State University
Raleigh, NC, USA
{mcgrace, wu_zhou, xuxian_jiang}@ncsu.edu

Ahmad-Reza Sadeghi
Center for Advanced Security Research,
Technical University Darmstadt
Darmstadt, Germany
ahmad.sadeghi@trust.cased.de

ABSTRACT

In recent years, there has been explosive growth in smartphone sales, which is accompanied with the availability of a huge number of smartphone applications (or simply apps). End users or consumers are attracted by the many interesting features offered by these devices and the associated apps. The developers of these apps benefit financially, either by selling their apps directly or by embedding one of the many ad libraries available on smartphone platforms. In this paper, we focus on potential privacy and security risks posed by these embedded or in-app advertisement libraries (henceforth "ad libraries," for brevity). To this end, we study the popular Android platform and collect 100,000 apps from the official Android Market in March-May, 2011. Among these apps, we identify 100 representative in-app ad libraries (embedded in 52.1% of the apps) and further develop a system called AdRisk to systematically identify potential risks. In particular, we first decouple the embedded ad libraries from their host apps and then apply our system to statically examine the ad libraries for risks, ranging from uploading sensitive information to remote (ad) servers to executing untrusted code from Internet sources. Our results show that most existing ad libraries collect private information: some of this data may be used for legitimate targeting purposes (i.e., the user's location) while other data is harder to justify, such as the user's call logs, phone number, browser bookmarks, or even the list of apps installed on the phone. Moreover, some libraries make use of an unsafe mechanism to directly fetch and run code from the Internet, which immediately leads to serious security risks. Our investigation indicates the symbiotic relationship between embedded ad libraries and host apps is one main reason behind these exposed risks. These results clearly show the need for better regulating the way ad libraries are integrated in Android apps.

Categories and Subject Descriptors K.6.5 [**Management of Computing and Information Systems**]: Security and Protection - Invasive Software

General Terms Security

Keywords Smartphone, Privacy, In-App Advertisement

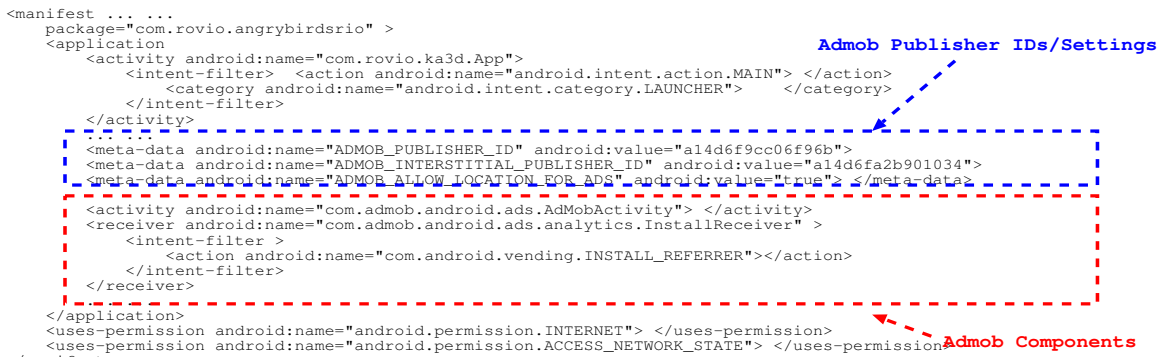
WiSec'12, April 16–18, 2012, Tucson, Arizona, USA.

1. INTRODUCTION

Over the past few years, smartphone sales have experienced explosive growth. According to Gartner, sales of these devices increased 74% year-on-year in the second quarter of 2011 [6] and late last year, smartphones already outsold the personal computers for the first time in history [25]. Evidently, the market has embraced these mobile devices due to their convenience and power: these sensor-rich devices are small enough to be carried like a traditional cellphone, yet offer their users a much wider range of functions than simple SMS messages or basic phone calls. Moreover, they are defined by the ability to download and run third-party apps that provide additional useful features. In other words, instead of being restricted to the functions provided by the phone manufacturers, carriers, or limited affiliates, smartphone users can partake of thousands of apps designed for purposes unforeseen by the parties involved in making and distributing the devices. Furthermore, platform vendors (e.g., Google and Apple) also provide centralized app markets where users can simply tap through the process of browsing, searching, purchasing, downloading, and installing these apps.

As part of the mobile eco-system, the app developers, largely motivated by financial incentives, submit their apps to centralized app markets for users to access. Notice that on the Android platform, almost two-thirds of all apps are free to download [5]. To be compensated for their work, many app developers incorporate an advertisement library (also known as an ad library) in their apps. At run-time, the ad library communicates with the ad network's servers to request ads for display and might additionally send analytics information about the users of the app. (For simplicity, we use the term ad libraries to represent both ad libraries and analytics libraries.) The ad network then pays the developer on an ongoing basis, based on metrics that measure how much exposure each individual app gives to the network and its advertisers.

In this paper, we aim to study existing in-app ad libraries and evaluate potential risks from them. Specifically, we focus on the Android platform and determine what risks the popular ad libraries on Android may pose to user's privacy and security. To this end, we collected 100,000 apps from the official Android Market in a three-month period, i.e., March-May, 2011. Among these apps, we identify and extract 100 representative ad libraries that are used in 52,067 (or 52.1%) of them. To facilitate our analysis, we further developed a static analysis tool called AdRisk to analyze the extracted ad libraries and report possible risks. In particular, our current analysis mainly focuses on those "dangerous" permissions (Section 2) defined in the standard Android framework, seeking to identify their possible (mis)use by ad libraries.




```
<manifest ... ...
    package="com.rovio.angrybirdsrio" >
    <application
        <activity android:name="com.rovio.ka3d.App">
            <intent-filter>  <action android:name="android.intent.action.MAIN"> </action>
                <category android:name="android.intent.category.LAUNCHER">    </category>
            </intent-filter>
        </activity>
        ... ...
        <meta-data android:name="ADMOB_PUBLISHER_ID" android:value="a14d6f9cc06f96b">
        <meta-data android:name="ADMOB_INTERSTITIAL_PUBLISHER_ID" android:value="a14d6fa2b901034">
        <meta-data android:name="ADMOB_ALLOW_LOCATION_FOR_ADS" android:value="true"> </meta-data>
        <activity android:name="com.admob.android.ads.AdMobActivity"> </activity>
        <receiver android:name="com.admob.android.ads.analytics.InstallReceiver" >
            <intent-filter >
                <action android:name="com.android.vending.INSTALL_REFERRER"></action>
            </intent-filter>
        </receiver>
        ... ...
    </application>
    <uses-permission android:name="android.permission.INTERNET"> </uses-permission>
    <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"> </uses-permission>
</manifest>
```

Figure 1: The (Abbreviated) *AndroidManifest.xml* File in the Popular Angry Birds Android App (*com.rovio.angrybirdsrio*)

Our analysis revealed a number of privacy and security issues in the 100 representative ad libraries. In particular, most ad libraries collect private information. While some of them may use these information for legitimate purposes (i.e., the user's location for targeted advertising), we noticed a few ad libraries invasively collect information, such as the user's call logs, account information or phone number. Such information can be used to deduce the true identity of the user, enabling more comprehensive tracking of the user's habits – at the cost of all pretense of privacy. One particular popular ad library (used in 4190 apps in out dataset) even allows a variety of personal information to be directly accessible to the advertisers, creating unnecessary additional opportunities for misuse. We also found out that some ad libraries will download additional code at runtime from remote servers and execute it in the context of the running app, opening up the opportunities for exploitation and abuse and making it impossible to ensure its integrity. In fact, we have confirmed one particular case that fetches and loads suspicious payloads. After the finding, we reported those infected apps (7 in our dataset) to Google and all of these apps have now been removed from the official Android Market. These results call for the need for additional mechanisms to regulate the behavior of ad libraries on Android.

The rest of this paper is organized as follows: Section 2 is an overview of the relevant portions of the Android framework. Section 3 explains the system design to assess the threat posed by ad libraries, while Section 4 contains the implementation and evaluation results. Section 5 considers the implications and limitations of our work, which is followed by a survey of related work in Section 6. Lastly, we summarize our paper in Section 7.

2. BACKGROUND

To understand how an ad library is embedded into an Android app, we will consider a popular app, i.e., Rovio's *Angry Birds*, as the example. Initially a paid iPhone app, *Angry Birds* moved to an ad-supported model when Rovio ported it to Android. The game is free to download, but ads are displayed periodically during play and while loading new levels; these ads generate $1 million a month in revenue for Rovio.

Since Rovio is not in the advertising business, the company turned to third-party advertising networks to monetize *Angry Birds* on Android. This is a common arrangement and natural choice for smartphone app developers. After registering some financial information with an ad network, developers receive a developer identifier and a SDK. The SDK's documentation includes instructions on how to use the included ad library. Ad libraries are designed to be embedded in the app that uses them, so the instructions include the necessary permissions required by the ad library; the developer must make sure the ad-supported app requests these permissions by making the necessary changes to its manifest file. Similarly, in order to be paid for the ads served by the app, the developer must make sure the ad library is furnished with their developer identifier.

Angry Birds' manifest file (included as Figure 1) provides a representative example of this arrangement. This particular version of *Angry Birds* contains Google's popular AdMob ad library, which pulls some of its control data from the manifest of its host app. Such data includes the crucial publisher identifiers, which are stored as the "`ADMOB_PUBLISHER_ID`" and "`ADMOB_INTERSTITIAL_PUBLISHER_ID`" meta-data values. Also in the manifest file, AdMob listens for package installation events by registering the `com.admob.android.ads.analytics.InstallReceiver` component, and defines its own Activity (screen) with `com.admob.android.ads.AdMobActivity` to display full-screen ads.

In general, ad libraries can be classified into three ad-oriented categories: mobile web libraries, rich media libraries, and ad mediators. Mobile web libraries are front-ends to web-based ad networks. Content is requested, delivered and displayed using standard web technologies, with very little interaction with the device's APIs. These libraries typically display only banner or text ads. In our study, we found over half of existing in-app ad libraries are of this type. Rich media libraries have a similar mission, but behave more like powerful platforms. Specifically, they provide feature-rich APIs for both app developers and advertisers. While they can display the simpler ad types, they can also support more advanced kinds such as active content (i.e., JavaScript), video, interstitial ads and the like. Although there are fewer ad libraries as rich media libraries than mobile web libraries, many of the most popular ones, including AdMob, are actually rich media ones. The third category, ad mediators, is different from the previous two by exposing a standard interface through which an app developer can interact with other ad libraries of the other two types. Since ad libraries often request similar information from the app developer in very different ways, these mediator libraries exist to make bundling multiple ad libraries in an app easier.

Our experience indicates that all three kinds of ad libraries tend to share some common characteristics. For example, they have user-interface code (to present their ads) and network code (to request ads from the ad network's servers). They are also designed to be tightly bundled with host apps. In this way, it becomes more difficult to disable the ad functionality or defraud the ad network. To the same end, some ad libraries heavily obfuscate their internal workings in an effort to discourage reverse engineering. AdMob again provides a representative example. Inside the AdMob ad library, only the classes, methods and fields described in the AdMob documentation have meaningful names; everything else has had its name changed to a letter of the alphabet. Moreover, all debugging information is stripped from all the classes in the package.

Protection level	Description
`normal`	Low-risk permissions granted to any package requesting them
`dangerous`	High-risk permissions that require user confirmation to grant
`signature`	Only packages with the same author can request the permission
`signatureOrSystem`	Both packages with the same author and packages installed in the `system` image can request the permission

Table 1: Permission Protection Levels in Android

At runtime, the embedded ad libraries execute together with the host app inside the same runtime environment – a Dalvik [4] virtual machine (VM), which is eventually instantiated as a user-level process in Linux. Different apps run in different Dalvik VMs, isolated from each other. The Dalvik VM is derived from Java but has been significantly revised (with its own machine opcodes and semantics) to meet the resource constraints of mobile phones. When an app is installed in Android, it is assigned its own unique user identifier (UID) – as Android relies on the Linux process boundary and this app-specific UID assignment strategy to achieve isolation or prevent a misbehaving or malicious app from disrupting other apps or accessing other apps' files. Unfortunately, this strategy does not separate host apps from the in-app ad libraries they contain, as those libraries inhabit the same Dalvik VM and execute with the same UID. In our example, AdMob could readily send the user's *Angry Birds* scores to Google.

The situation is further complicated by the fact that Android apps are structured differently than programs on most platforms, in that they can contain multiple entry points. These entry points are invoked by the framework in response to inter-process communication (IPC) events; even "running" an app is treated in this way. Technically, each app is composed of one or more different components, each of which can be independently invoked. There are four types of components: *activities*, *services*, *broadcast receivers* and *content providers*. An activity represents part of the visible user interface of an app. A service, much like a Unix daemon, runs in the background for an indefinite period of time, servicing requests. A broadcast receiver receives and reacts to broadcast announcements, while content providers make data available to other apps. Each Android app is deployed in the form of a compressed package (`apk`). These `apk` files contain a manifest file (`AndroidManifest.xml`) that describes various standard properties about the app, such as its name, the entry points (or interfaces) it exposes to the rest of the system, and the permissions it needs to perform privileged actions. The *Angry Birds* manifest (Figure 1) describes two entry points defined by AdMob instead of *Angry Birds*: an activity (`com.admob.android.ads.AdMobActivity`) and a broadcast receiver (`com.admob.android.ads.analytics.InstallReceiver`). The activity is designed to be invoked by the code in the app, but the broadcast receiver is interested in `com.android.vending.INSTALL_REFERRER` events sent out by the Android Market app. Accordingly, it's possible to invoke the ad library's code directly before any of the host app's code is run.

To better protect personal information and manage system resources, Android defines a permission-based security model [2]. In this model, the principals that have these permissions are apps, not users or libraries. The Android framework contains a predefined set of permissions and also allows developers to define additional permissions as they see fit. Each permission has a protection level [1], which determines how "dangerous" the permission is and what other apps may request it. Table 1 summarizes the defined protection levels in Android. The `signature` and `signatureOrSystem` permission protection levels are reserved to define capabilities that are not meant to be used by apps written by other authors or by apps that are part of the system image. Permissions are checked either through annotating entry points defined in the manifest file or programmatically by the Android framework. Since ad libraries are not principals, they inherit the permissions of the apps they are embedded in. As a result, many ad libraries opportunistically check for and use permissions. Some may allow the host app's author to control their behavior somewhat while most ad libraries simply use what permissions granted to the host apps.

3. SYSTEM DESIGN

The goal of this work is to assess possible privacy and security risks posed by the embedded in-app ad libraries and additionally quantify these risks by measuring their prevalence on Android. Note that the Android's permissions-based security model provides a convenient way to measure the risk inherent in Android APIs, as their documentation typically mentions whether a permission check is required to successfully make the call. However, as mentioned previously, ad libraries are not annotated in any way by the Android framework. Also, the context surrounding each potentially-dangerous Android API call is very important in matters of privacy. For instance, if the user's phone number is retrieved but never sent to the Internet, no privacy violation has occurred. In this work, we opt to crawl and collect available apps from the official Android Market. After that, we systematically identify representative ad libraries from these apps and then develop a system to thoroughly identify possible risks. Figure 2 summarizes the methodology in our study.

3.1 Sampling the Android Market

We crawled the Android Market for apps over three months (March through May, 2011) and chose the first 100,000 downloaded apps as the dataset for our study. With them, we built a database that extracts the features needed to perform our later analysis, i.e., the permissions requested by each app (as defined in its manifest file) as well as the Java class tree hierarchy contained in the app's code.

After that, among the 100,000 apps, we select apps that have the `android.permission.INTERNET` permission, which is required for communication with the ad network's servers, and organize them into a candidate set. From the candidate set, we randomly select an app and disassemble it. The disassembled bytecode is examined for new ad libraries. Especially, in the search process for new ad libraries, we maintain an ad set, which is initialized to be empty. For each new ad library we identified, we add it to this set. Further, we extract its unique class tree and use it as the pattern to detect the list of host apps that contain this particular ad library. Specifically, we remove those host apps from the candidate set. We repeat the selection process until 100 distinct ad libraries have been selected. By searching the class trees stored by the database for each ad library's package name, we can then determine how many apps within our sample of 100,000 contain the given ad library. Sorting and graphing these figures of the top 20 ad libraries produces the graph in Figure 3. (The list of 100 ad libraries is detailed in Tables 2 and 4 – Section 4.) In total, the 100 ad libraries in our study are present in 52.1% of the collected 100,000 apps.

Among these 100 representative ad libraries, Google's own AdMob, AdSense, and Analytics networks are listed in the top five. We also note that several other networks – Flurry, MillenialMedia, Mobclix, and AdWhirl – appear in a comparatively large number of apps. Given the maturity of these ad networks behind these lead-

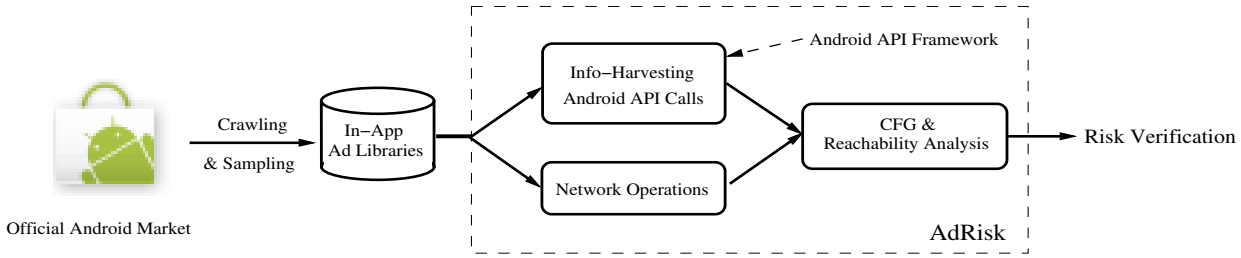


Figure 2: Assessing Possible Risks in Smartphone In-App Advertisements

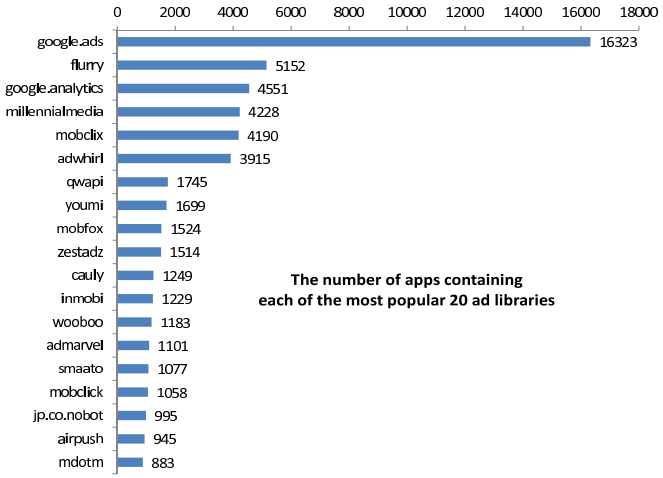


Figure 3: Popularity of the Top 20 Ad Libraries in Our Study

ing libraries, we expect that the libraries themselves offer standard functionality and do not engage in activities frowned upon by the industry as a whole. On the other hand, any potential privacy risks posed by such commonly-deployed libraries would impact many users. Among the remaining libraries, there allegorically appear to be a large number of small ad networks that offer in-app ad libraries on Android. The large number of such libraries, coupled with the relatively small proportion of apps they appear in, make holding their behavior to account more difficult for watchdog organizations inside and outside the ad industry. Analyzing these libraries is therefore important in order to gain perspective on the range of behaviors ad networks will engage in.

3.2 Analyzing Ad Libraries

After identifying the 100 representative ad libraries, we next seek to determine whether a given ad library contains any risks to security or privacy. To do that, we start by considering the permission protection levels [1] defined by the standard Android framework. Note that various standard APIs exposed by the framework require certain permissions to access, which have been annotated by a protection level. Any APIs that require a permission with an elevated protection level (i.e., above "normal") can be considered a risk to security or privacy.

Unfortunately, the relationship between APIs and permissions can be difficult to determine. The Android documentation does not feature an exhaustive list of these relationships, and some permissions are only conditionally checked. For example, Android defines two related permissions that allow access to the user's location data: `android.permission.ACCESS_COARSE_LOCATION` and `android.permission.ACCESS_FINE_LOCATION`. Both permissions are checked by the methods of the `android.location.LocationManager` class; however, these methods determine which permission to check by the arguments they are given. For example, calling `LocationManager.getLastKnownLocation("gps")` requires the `android.permission.ACCESS_FINE_LOCATION` permission; the same call with the argument of `"wifi"` would instead require the `android.permission.ACCESS_COARSE_LOCATION` permission.

To address these challenges, we apply and extend Felt et al. [16] to derive a list of API calls that are of interest for our analysis. In particular, we take a similar approach by analyzing the Android documentation, source code and disassembled bytecode to conservatively annotate the standard APIs with the permissions that they require. However, unique to this study, our extensions also include a new set of Android API calls, which do not require any permission (Section 4). In particular, most of them are related to `ClassLoader` and reflection mechanisms. The `ClassLoader` part is responsible for dynamically loading code at runtime. To elaborate, in Dalvik, class references are resolved at run-time. Usually, due to the presence of a verifier looking for undefined references, it is safe to consider a Dalvik app as containing only well-defined static code. When combined with reflection API, it becomes possible to refer to classes using data at run-time, thus invoking the `ClassLoader` functionality *after* the verifier has run. Since the `ClassLoader` is itself just a class, its methods can be overridden to allow developers to pass raw bytecode to the Dalvik VM at runtime. In this fashion, it is possible to download and run arbitrary dynamic code, rendering any static analysis of an app incomplete. Fortunately, the interfaces to the underlying Dalvik VM are well-defined. We treat these interfaces as just another kind of APIs, which not only implicitly marks dynamic code loading as a suspicious behavior, but unifies our analysis framework. In total, our current system considers 76 distinct permissions (34 `dangerous`, 26 `signature`, 11 `signatureOrSystem`, and 5 `normal` – Section 4).

3.3 Identifying Possible Risks

After identifying the set of APIs of interest, we then perform a reachability analysis for each ad library. We are interested in two dimensions of potentially dangerous behaviors, which means we must deal with up to four potential reachability conditions. The first dimension involves the precipitating event for the dangerous behavior; that behavior could come from one of Android's many entry points, or could be in response to a received network packet. Finding a path from either of those start points to an API *could* signal a dangerous situation, but may not necessarily; this is where the second dimension comes into play. Some API calls are dangerous in themselves (such as those that can cost money) while others merely expose personal data that can then be leaked to an external party. In the first case, finding a path from an initiating entry point or network connection is sufficient, but in the second we must further find a dataflow path from the dangerous call to an external sink (e.g., network APIs).

In mechanical terms, our method is as follows: each ad library sample's bytecode is first scanned for the dangerous API calls we previously annotated. For each found API call, we trace backwards through the library source looking for potential entry points and any mitigating circumstances; for example, if such a dangerous API call only occurs if a flag representing the user's consent is set, we note this behavior. Some API calls may not be reachable under any circumstances and therefore may be safely ignored, but all others are recorded if they match these conditions. For those calls that leak information, we then additionally trace forwards through the bytecode looking for a network sink. If one is found, the candidate path from the API call to the network is also recorded[1]. In algorithmic terms, we produce a control-flow graph showing all the possible paths of execution through the library, then determine which of those paths are indeed feasible.

In our prototype, we leverage the existing `baksmali` Dalvik disassembler [3] to automate some of this process. As part of the greater `smali` package, this allows us access to a convenient intermediate representation and a limited set of intra-procedural static analysis tools. Using it as the base, we add code to derive the control-flow graph which we will traverse to find the set of feasible paths through the app (and thus the ad library).

Due to a key difference between Android apps and traditional Java programs, traversing the derived control-flow graph poses additional challenges under Android. Specifically, Java programs, like those written in many other languages, start execution at a `main` method. Android apps have no such method, instead containing a number of entry points based on the components they contain (certain methods in e.g., `Service` and `Receiver` objects). In addition to these, the library itself usually exposes some methods to the host app for initialization purposes. The entry points specified by the framework are automatically identified on the basis of the class they belong to, while the library's initialization methods are fed into the system through annotation. We then run the subsequent steps in our analysis over each entry point in turn, finally merging the results.

Our experience indicates that due to the influence of native code and the core classes in Android framework (e.g., the use of threads – a common technique in Android for improved user responsiveness), we observe discontinuities in the generated call graph. To resolve these discontinuities, we elect to load an additional set of class files alongside the library. These files stand-in for core classes and contain simple expressions designed to capture the semantics of each API call. Additionally, these files include the dangerous API calls the system is supposed to identify; each dangerous call contains a sentinel instruction that alerts our analysis code to its nature for the next stage of analysis.

Given this control-flow graph, our algorithm next attempts to find reachable paths from an entry point to a dangerous API call. To do this, we perform the traditional information-flow analysis, where constraints are placed on the variables and checked against by branch instructions. In the resulting feasible control-flow graph, we verify whether each dangerous API call is in a feasible code region. If a call is, execution is traced backwards and the necessary constraints remembered to form an execution path, which is then reported. The paths reported by our system are then verified. The reentrant, multi-threaded nature of Android apps makes points-to analysis difficult, which in turn frustrates efforts to accurately identify only feasible paths through the library. Certain language features are not fully supported yet in our current prototype. For example, the Java Reflection APIs (i.e., the `java.lang.reflect.*` package) allow code to be invoked by name, and without perfect dataflow analysis tools this causes an irreconcilable discontinuity in the generated control-flow graph. To accommodate such situations, we take a conservative approach wherever possible, preserving accuracy but necessitating additional manual effort on some occasions. In particular, we report the use of reflection APIs in an ad library to highlight their presence for further investigation (Section 4).

[1]Note that we do not ignore calls that do not meet this additional criterion due to the complexities inherent in dataflow analysis. It is possible to introduce dataflow discontinuities using threading, caching, and other behaviors; we elect to involve some additional manual effort in order to ensure the accuracy of our study.

4. PROTOTYPING AND EVALUATION

We base our static analysis tool on the open-source `baksmali` Dalvik disassembler (version 1.2.6). Implementing the design laid out in the previous section required 2809 new lines of code and four hooks in the original `baksmali` project. As stated in the design section, our system also required each API of interest to be annotated so that it could be analyzed by the system. Accordingly, we annotated APIs associated with 76 standard Android permissions. As our static analysis approach is rather standard, in the following, we mainly focus on the peculiarities of the Android platform and the new extensions we added for risk analysis.

Specifically, besides reporting potentially-feasible paths, our prototype has been extended to report on five other code patterns of interest: the use of reflection, dynamic code loading, permission probing, JavaScript linkages and reading the list of installed packages. As the presence of one or more of these patterns can color our other findings for a given library, we opt to have our tool automatically report them alongside its feasible-path output.

The first such pattern, the use of reflection, concerns the use of the `java.lang.reflect` package. As mentioned earlier, this portion of the Java specification allows programmatic invocation of methods and access to fields, which complicates our static analysis. Without it, the static analysis of Dalvik bytecode is reliable and unambiguous. In theory, reflection essentially makes resolving an app's call graph into a dataflow problem. In practice, often reflection appears to involve constant strings, thus introducing no new ambiguity. However, this is not always the case in our collected ad samples. Therefore, our system makes what assumptions it can while flagging the situation for further review.

In a similar vein, Android apps usually are amenable to static analysis techniques because they are designed to be loaded as a whole and statically verified by the framework itself. However, another esoteric Java language feature was carried over into Dalvik: the `ClassLoader` class. This class is used by the framework to find code resources on demand. Usually, the static verification stage causes practically the entire app to be loaded at once, as the verifier attempts to resolve all the references in the bytecode. However, using reflection, it is possible to cause a class to be loaded that is not directly referenced by any existing code. As the `ClassLoader` class can be extended by developers, custom versions of this class can be written to load code from non-standard resources. Each such `ClassLoader` inherits from a parent version of the class, on up to the baseline "system" instance of the class. Since Dalvik, unlike Java, does not permit the "system" `ClassLoader` to be changed by the developer, dynamic code loading is very explicit: the generic reflection API cannot be used to implicitly reference a class for the first time, so instead the custom `ClassLoader` must be explicitly queried. Our prototype flags this behavior and raises a serious warning, as its presence negates all existing static analysis efforts and signals suspicious dynamic code loading behavior.

A more common pattern our prototype elects to handle specially is what we call "permission probing." In this pattern, an ad li-

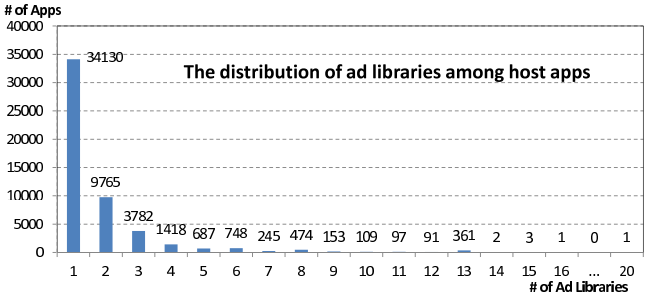


Figure 4: Number of Ad Libraries Contained by Each App

brary contains some API which requires permission to successfully call. Instead of mandating that the developer of the host app requests this permission, the ad library can instead opportunistically attempt to use the API, either by checking that it has the necessary permission beforehand, or by handling the `SecurityException` that is thrown by most APIs when they are called with insufficient permission. These methods of checking permissions are well-defined under Android, and so we can inspect the control-flow graph to detect branches that detour around dangerous API calls.

Similarly, it seems to be common practice for "rich media" ad libraries to offer JavaScript bindings to expose additional functionality to JavaScript ads. We elect to include this practice in our findings for two reasons. Not only is this behavior indicative of "rich media" libraries, it also raises interesting privacy concerns, which we will cover in greater depth in Section 4.2.2.

Lastly, we temper our results by showing one instance of an invasive API that, for whatever reason, requires *no* special permission to access. Some ad libraries we studied collect the list of all apps installed on the device. This information is every bit as personal as the user's browser history, in that it reveals some information about their interests. We include this behavior to demonstrate the guile advertisers have and the incompleteness of the permission-based system.

In the rest of this section, we present our findings from the analysis of 100 representative ad libraries. We first summarize our results in Section 4.1 and then present categorized findings about dangerous behaviors in these ad libraries in Section 4.2. Finally, we evaluate the performance of our prototype in Section 4.3.

4.1 Overall Results

Before tabulating our findings, we stress that our results are for ad libraries rather than apps. Some apps in our sample of 100,000 will contain more than one ad library, while others contain none at all. For the 100 representative ad libraries in our study, we found that they are embedded in 52,067 different host apps in our dataset. As one host app may contain more than one ad library, we show a breakdown of how many libraries each of these apps contain. The result is shown as Figure 4.

From the figure, it seems more than one third of apps (or more precisely $35,991$) contain one ad library and a small fraction of apps (around 3%) include at least five ad libraries for monetization. One particular host app, i.e., `com.Dimension4.USFlag`, embedded no fewer than 20 ad libraries! However, it is unclear whether the inclusion of more ad libraries necessarily brings more profit to an app's developers.

Our system scans each representative ad library for the use of 76 dangerous APIs. The overall results for the top 50 ad libraries are shown in Table 2, while the results for the remaining 50 are shown in Table 4 (Appendix). In practice many of the Android's dangerous APIs were not used by any ad library, so we choose to omit them for brevity in our results. Specifically, the two tables contain the 14 dangerous APIs we see used by at least one ad library. In the tables, we also include data on six structural properties of interest, such as the use of obfuscation, conditional API use via permission probing, and dynamic code loading through the `ClassLoader` language feature. Overall, our system reports 318 total API uses and structural patterns. Upon further verification, 19 of them ask permission from the user and our system properly recognizes 15 of these cases, which happen to be all related to text message (SMS) API calls.

Despite all of the reported APIs being marked as "dangerous," our results show that some APIs are commonly used by ad libraries. These include the location APIs and a single "Read Phone Information" API call, both of which are used by at least half of the ad libraries we analyzed. The ad libraries use these APIs for targeting information: the location APIs can be used to serve ad content relevant to the geographic area of the user, while the commonly-used phone information call returns a unique identifier (the phone's IMEI number) that is useful for tracking what content has been served to a particular user. These uses seem plausible, in the context of an ad library; however, we did identify two ad libraries (Mobclix and adserver) that expose this information directly to advertisers, which is harder to justify.

The remaining dangerous APIs either provide some feature, or allow access to more intrusive data maintained by the device. The feature-based APIs appear to be mostly harmless. For example, a number of ad libraries allow ads to place phone calls, send text messages or add an event to the calendar. In all of these cases, these functions are performed only after the user triggers them (i.e., by clicking on an ad) *and* confirms their intentions.

More insidious, however, are requests for information that is not directly useful for ad targeting. Our analysis uncovered a few instances where an ad library accessed information that is only useful when correlated with other facts known about the user. This correlative information is a direct threat to the user's privacy, because it can be used to uncover the user's true identity. For example, it is hard to make a case that the user's call history has any bearing on what ads they will be interested in, yet we discovered one ad library (sosceo) transmitting some of that information to the Internet (to be detailed in Section 4.2.1). In a similar vein, a large number of ad libraries used an API call to retrieve the user's phone number, and another ad library (Mobus) peculiarly reads through the user's SMS messages to determine which text-messaging service center they use. Finally, we identified one particularly worrying use of an otherwise innocent API, where some ad libraries (such as waps) upload a list of all the installed apps on the phone.

Looking beyond privacy concerns, we identified five ad libraries which make use of the `ClassLoader` feature to dynamically load code at runtime. These ad libraries are effectively impossible to statically analyze as a result; at a whim, their code can be changed. A malicious or compromised ad network could command its ad libraries to download a botnet payload or root exploit, for example. Our later investigation indeed captures one suspicious payload, which essentially turns the host app into a remotely-controllable bot (Section 4.2.3).

Moreover, the other structural properties of ad libraries are worth mentioning. Over half of the ad libraries we studied employed obfuscation techniques, presumably to discourage reverse engineering. While not altering the function of the library, these transformations strip human-readable names from methods and classes while optionally muddling the control flow by adding pointless redundancy or by reordering instructions. As an example, we list in Figure 5 the classes contained in one particular ad library, i.e., AirAD. Only a few such classes have names that carry any meaning; all the

		Included in Apps	Probes Permissions	Uses Obfuscation	Uses Reflection	Uses JavaScript	Read Installed Packages	Location Data	Place Phone Call	Camera	List Accounts	Read Calendar	Read Contact/Call Logs	Read Browser Bookmarks	Read Phone Information	Read Phone Number	Read SMS	Send SMS	Change Calendar	Change Contacts	Use Vibrator	ClassLoader
admob/android/ads	27235	✓	✓	✓	✓	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
google/ads	16323	✓	·	✓	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
flurry	5152	✓	✓	·	✓	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
google/../analytics	4551	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
millennialmedia	4228	✓	·	·	✓	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	✓	·
mobclix	4190	✓	·	✓	✓	·	✓	·	·	✓	✓	✓	·	✓	·	·	·	✓	✓	✓	·	·
adwhirl	3915	✓	·	✓	·	·	✓	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
qwapi	1745	✓	·	·	·	·	✓	·	·	·	·	·	✓	✓	·	·	·	·	·	·	·	·
youmi	1699	✓	✓	·	·	·	✓	·	·	·	·	·	✓	·	·	✓	·	·	·	·	·	·
mobfox	1524	✓	·	·	·	·	✓	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
zestadz	1514	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
cauly	1249	·	·	·	·	·	✓	✓	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
inmobi	1229	✓	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
wooboo	1183	✓	✓	·	·	·	✓	·	·	·	·	·	✓	✓	·	✓	·	·	·	·	·	·
admarvel	1101	✓	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
smaato	1077	✓	·	·	✓	·	✓	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
mobclick	1058	✓	·	·	·	·	✓	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
jp/co/nobot	995	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
airpush	945	✓	✓	·	·	·	✓	·	·	·	·	·	✓	·	·	✓	·	·	·	·	·	·
mdotm/android/ads	883	·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
vdopia	872	·	✓	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
wiyun	777	✓	✓	✓	·	·	✓	✓	·	·	·	·	✓	·	·	✓	·	·	·	·	·	·
android/adhubs	651	·	·	·	✓	·	✓	·	·	·	·	·	✓	✓	·	·	·	·	·	·	·	·
madhouse	603	✓	✓	✓	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
pontiflex	522	✓	·	✓	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
innerActive	497	✓	·	·	·	·	·	✓	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
adserver/adview	492	✓	·	✓	✓	·	✓	✓	·	·	✓	·	✓	·	·	·	·	✓	·	·	·	·
casee	479	✓	·	·	·	·	✓	✓	·	·	·	·	✓	✓	·	·	·	·	·	·	·	·
greystripe	440	·	✓	·	✓	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
omniture	433	·	·	✓	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
guohead	400	✓	·	✓	✓	✓	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
daum/mobilead	399	✓	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
domob	374	✓	✓	·	·	·	✓	·	·	·	·	·	✓	·	·	✓	·	·	·	·	·	·
tapjoy	368	·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
jp/Adlantis	341	·	·	✓	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
adagogo	339	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
adchina	327	·	·	·	·	·	·	✓	·	·	·	·	✓	✓	·	✓	·	·	·	·	·	·
jumptap	278	✓	·	·	✓	·	✓	✓	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
medialets	274	✓	✓	✓	✓	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	✓	·
nowistech	272	·	·	·	·	·	·	·	·	·	·	✓	✓	·	·	·	·	·	·	·	·	·
waps	239	✓	·	✓	·	✓	✓	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
vpon/adon	189	·	·	·	·	·	✓	·	·	·	·	·	✓	·	·	✓	·	·	·	·	·	·
energysource	160	✓	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	✓
iconosys	131	·	·	·	·	·	·	·	✓	✓	·	✓	·	✓	✓	·	✓	·	·	·	·	·
adwo/adsdk	131	✓	✓	·	✓	·	·	·	·	·	·	·	✓	✓	·	✓	·	·	·	·	·	·
sktelecom/tad	125	✓	✓	✓	✓	·	✓	✓	·	·	·	·	✓	✓	·	·	·	·	·	·	·	✓
kr/uplusad	112	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
smartadserver	102	·	·	✓	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
mt/airad	89	✓	✓	·	·	·	✓	·	·	·	·	·	✓	✓	·	·	·	·	·	·	·	·
emome/hamiapps/sdk	85	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
Total	92297	31	14	15	17	3	27	8	1	2	2	3	0	33	9	0	9	2	1	3	2	

Table 2: The Overall Results from the Top 50 Representative Ad Libraries

rest are strings of "l" (lowercase L) and "I" characters. The ad libraries that are noted as using obfuscation in Table 2 all used some scheme to obfuscate their internal classes, and typically also obfuscate the names of fields, methods and the like in a similar fashion. Other obfuscators are known to exist, but all serve the same purpose; for example, the default obfuscator names classes after alphabetical characters, while another uses nonsense dictionary words like "Watermelon" and "Railroad." Applying these techniques to a reasonably large ad library hides the intent behind much of what the library does, while not truly protecting the ad network's trade secrets – as the library can still be unambiguously analyzed, and the network's core functionality resides on its servers regardless, safe from competitors' eyes.

In another common pattern, many ad libraries probe the permissions available to them before attempting to use permission-guarded APIs. Normally, if an Android app calls an API it does not have permission to access, a `SecurityException` is thrown. If this exception is not caught, the app will crash. In order to prevent this from happening, ad libraries either check their permissions up front or silently catch the thrown exception. It turns out more than half of studied ad libraries (marked in Tables 2 and 4) engage in this sort of behavior. Some of them do log their failed attempts to access

Figure 5: Classes in the Obfuscated `com.mt.airad` Package

these APIs, chastising the host app's developer for not properly requesting the necessary permissions. However, most attempt to do what they can with as many permissions as they can access, again silently. A few libraries, such as AdMob, do permit the host app developer to selectively deny the library permission to use a certain API. This is unfortunately far from the norm, and only Mobclix allows the *user* to disallow access to sensitive APIs – on a case-by-case basis, and with some troubling ramifications, as elaborated in Section 4.2.2.

Lastly, some ad libraries use the Java reflection language feature, which essentially allows programmatic access to methods and fields by their name. Normally, when Dalvik bytecode is loaded, there is a static verification step that ensures all referenced code elements are valid. Reflection sidesteps this mechanism, which allows for the use of dynamic code (discussed at more length in Section 4.2.3), but can also be used to access *any* code that is not guaranteed to resolve correctly on all devices. In this way, it is possible for ad libraries to access "experimental" APIs or vendor-specific APIs. Given the lower maturity of such APIs, their use by ad libraries is suspect.

4.2 Categorized Findings

To provide greater detail about problematic behaviors we came across in our analysis, we organize them into three categories.

4.2.1 Invasively Collecting Personal Information

The first category involves the questionable collection of personal information. Specifically, some ad libraries brazenly request information not directly useful in fulfilling their purpose. Our results show that the larger ad networks typically do not engage in such questionable activities, but smaller ad networks might. Unfortunately, there is no way for the user of an app to know which ad networks it contains.

A representative example of this behavior can be found in the sosceo ad library, one of the least popular libraries studied. Like most ad libraries, sosceo is instantiated by its hosting app making a UI element designed to display an ad, in this case a `com.sosceo.android.ads.AdView` object. When this object is created, a fairly lengthy set of obfuscated method calls occurs. These method calls ultimately query the device's contact information database for the most recent phone call. This information is duly stored in a field of a data object used by the `AdView` object; when the `AdView` object requests an ad from the backing ad network, the information is included as an URL query string under the "dp" key to the ad server.

Other ad libraries engage in similarly strange behaviors; Mobus, for example, reads the SMS (text-message) database looking for administrative information about the user's Short Message Service Center (SMSC). This SMSC is the back-end service provider responsible for routing text messages to and from the user. For some unknown purpose, Mobus transmits this information to its servers.

Similarly, Pontiflex takes an interest in what account credentials the user has on the device. This information is not a direct security risk as the ad library does not have access to the credentials themselves, but it does query the list of accounts the user's phone manages. Somewhat suspiciously, the dangerous API calls in this case are performed via the reflection API, which is a language feature that allows methods to be invoked by means of data strings. It is possible that reflection is being used, in this case, to throw off static analysis of the library.

4.2.2 Permissively Disclosing Data to Running Ads

The second category involves the direct exposure of personal information to running ads. One of the most popular ad libraries, Mobclix, appears at first glance to function like most other ad libraries. To display an ad, Mobclix creates an `android.webkit.WebView`, which is essentially a miniature web browser. The ad is then rendered by this web browser for display, allowing the advertiser to design their ads using standard web technologies.

However, unlike its principal competitors, Mobclix attempts to gain advantages by offering its advertisers access to certain smartphone features. Since these features do not have standard hooks in HTML or JavaScript, the Mobclix ad library has a class (`com.mobclix.android.sdk.MobclixJavascriptInterface`) that binds certain Android APIs to JavaScript functions that are then exposed to ads rendered within the `WebView`. Each API call is wrapped in a method that simply and succinctly exposes it to JavaScript. By doing so, Mobclix exposes a great variety of API calls and allows running ads to most of the sensors and data on the phone. Note that most of these accesses include appropriate user confirmation dialogs. For example, while an ad can call `contactsAddContact(...)` to add a contact to the user's address book, nothing will happen unless the user gives consent via a dialog box.

Unfortunately, not all functions are safely wrapped in this way. For instance, the `gpsStart(...)` function allows a JavaScript ad to register a callback function. This function will be called immediately, and again whenever the user's moves more than a defined distance from the last reported position. The user is never asked for their consent, nor are they notified in any way that this feature of their phone is being used by an ad. This particular example sufficiently raises interesting privacy issues. It is reasonable to expect that the Mobclix ad library itself should have access to location information; such information is commonly used to target ads to a certain geographical area. However, this code is not actually using that information for Mobclix's ad-targeting purposes. Instead, the information is being given to a third party advertiser. Indeed, given access to this functionality, the ad itself can be thought of as dynamically-loaded code of unknown provenance (Section 4.2.3).

4.2.3 Unsafely Fetching and Loading Dynamic Code

The third category involves unsafe fetching and loading of dynamic code (possibly from the Internet), which poses an even greater potential threat for two reasons. One is that this dynamically loaded code cannot be reliably analyzed, effectively bypassing existing static analysis efforts. The other is the fact that the downloaded code can be easily changed at any time, seriously undermining the capability of predicting or confining its behavior.

In the 100 representative ad libraries, five of them have this unsafe practice. One particular one will be downloading suspicious payloads, which allows the host app to be remotely controlled. Specifically, the portion of this ad library that is embedded in the host app is very small: a single service, `com.plankton.device.android.service.AndroidMDKService`. This service contacts a remote server with the list of permissions granted to the host app and the phone's hardware identifier (IMEI); in return, the remote server provides it with the URL to download a `.jar` file (see Fig-

```
POST /ProtocolGW/installation HTTP/1.1
Content-Length: 1242
Content-Type: application/x-www-form-urlencoded
Host: www.searchwebmobile.com
Connection: Keep-Alive

action=get&applicationId=123456789&developerId=987654321&deviceId=354957034053382
&currentVersion=-1&permissions=android.permission.INTERNET%3Bandroid.permission.
ACCESS_WIFI_STATE%3Bcom.android.browser.permission.WRITE_HISTORY_BOOKMARKS%3B
com.android.browser.permission.READ_HISTORY_BOOKMARKS%3Bcom.android.launcher.
permission.INSTALL_SHORTCUT%3Bcom.android.launcher.permission.UNINSTALL_SHORTCUT%3B
com.android.launcher.permission.READ_SETTINGS%3Bcom.android.launcher.permission.
WRITE_SETTINGS%3Bcom.htc.launcher.permission.READ_SETTINGS%3Bcom.motorola.launcher.
permission.READ_SETTINGS%3Bcom.motorola.launcher.permission.WRITE_SETTINGS%3Bcom.
motorola.launcher.permission.INSTALL_SHORTCUT%3Bcom.motorola.launcher.permission.
UNINSTALL_SHORTCUT%3Bcom.motorola.dlauncher.permission.READ_SETTINGS%3Bcom.motorola.
dlauncher.permission.WRITE_SETTINGS%3Bcom.motorola.dlauncher.permission.INSTALL_SHORTCUT
%3Bcom.motorola.dlauncher.permission.UNINSTALL_SHORTCUT%3Bcom.lge.launcher.permission.
READ_SETTINGS%3Bcom.lge.launcher.permission.WRITE_SETTINGS%3Bcom.lge.launcher.permission.
INSTALL_SHORTCUT%3Bcom.lge.launcher.permission.UNINSTALL_SHORTCUT%3Bandroid.permission.
READ_CONTACTS%3Bandroid.permission.READ_PHONE_STATE%3Bandroid.permission.READ_LOGS%3B

HTTP/1.1 200 OK
Date: Sun, 05 Jun 2011 04:30:33 GMT
Server: Apache-Coyote/1.1
Content-Length: 76
Connection: keep-alive

url=http://www.searchwebmobile.com/ProtocolGW/;fileName=plankton_v0.0.4.jar;
```

Figure 6: Handshake Communication between Plankton and its Command-and-Control Server

ure 6). This `.jar` file contains the vast majority of Plankton's code, which is then dynamically loaded using a `dalvik.system.DexClassLoader` object – Dalvik's base implementation of the `ClassLoader` Java language feature. The downloaded `.jar` will listen to remote commands and turn the host app into a bot. Based on this discovery, we have reported the seven affected host apps to Google, which promptly removed them from the official Android Market on the same day.

This behavior is interesting because it highlights the dynamically-linked nature of Dalvik. Android apps are distributed as bytecode, which makes app analysis easier due to the clearly-defined semantics of the format. Furthermore, upon loading a class for the first time, a Java-style bytecode verifier makes certain that all references within the class resolve. This verification step seems to preclude adding arbitrary code at runtime. However, via the `java.lang.reflect` package, Java (and hence Dalvik) can load classes by name at runtime. Coupling this language feature with the ability to control where Dalvik looks for definitions for such classes – that is, the `DexClassLoader` class – allows apps to load arbitrary code not contained in the app's package file. In this case, the downloaded `.jar` file has a predefined entry point, `com.plankton.device.android.AndroidMDKProvider.init(...)`. `DexClassLoader` looks for it by name and then invokes the control logic. Within the newly-downloaded code, the bytecode verifier works as usual, since it now uses the modified `DexClassLoader` to resolve references to unfamiliar classes.

Another four ad libraries make use of this feature, likely as a version-control and content-delivery mechanism. Opening the full expressive power of Dalvik – replete with all the permissions granted by the app – to nebulously downloaded dynamic code has unfortunate privacy implications. Again, given that the code retrieved from the Internet will naturally change, it is impossible to verify that the ad library is only engaging in the behaviors embodied in the library.

4.3 Performance Measurement

Next, we report the performance overhead of our prototype. In our test, we picked up five ad libraries and run our system to analyze each of them ten times. Each analysis run scans the given ad library for all (80) APIs our prototype handles. In each run, we record the processing time and report the average. Our test machine is an AMD Athlon 64 X2 5200+ machine with 2GB of memory and a Hitachi HDP72502 7200 rpm hard drive. We summarize the results in Table 3. The test-case libraries were selected to provide a mix of ad library types and complexities. Each library took, on average, $\sim$ 15.66 seconds to process. Given our tool is designed to be used in an offline, semi-automated capacity, we believe this performance to be acceptable for our purposes.

Library	Processing Time
AdMob	16.17s
AdWhirl	17.25s
Appmedia	14.58s
Quattro	14.40s
UplusAd	15.91s

Table 3: Processing Time of Analyzed Ad Libraries

5. DISCUSSION

Our study has so far uncovered a number of serious privacy and security risks from existing in-app ad libraries on the popular Android platform.[2] Given this, it is important to examine possible root causes and explore future defenses.

First, due to the fact that ad libraries are incorporated into the host apps that use them, they in essence form an symbiotic relationship. Based on such relationship, an ad library can effectively leverage it and naturally inherit all permissions a user may grant to the host app, thus undermining the app-based privacy and security safeguards. Accordingly, we believe that the exposed risks are fundamentally rooted in the granularity problem in the essential Android's permissions model. Under this model, the smallest entity that can be granted a permission is an app. Even though ad libraries come from a different developer and have different intentions than their hosting apps, they are afforded the same permissions. As we have seen, advertisers themselves are sometimes allowed to execute code within an app, adding yet another untrusted set of principals to the list of parties covered by a single permissions policy. Though an app's requests for access to private information can stem from the app's code, the ad library's code, or both, but the user or rather the Android platform cannot determine at a glance which parties will use the information.

Second, the current situation could also be a product of one central tension: the same solutions that would allow ad libraries to be sandboxed could also be used to disable them, or alternatively, defraud them. Even if Google had provided a separate Advertiser template in the Android framework (i.e., alongside the Services, Receivers, ContentProviders and Activities that exist today), there would be no incentive for ad networks to use it. It is safer to tightly couple ad libraries with their host apps, to keep them from being easily circumvented. Possibly for the same reason, some ad networks take the approach of the worrisome dynamic code loading behavior we observed. In particular, since ad libraries are not their own entity in the framework, they can only be updated alongside their host app. The ad network cannot control the release schedule of all the apps its ad library is bundled with. As a result, any code updates need to be pushed out along side channels. The dynamic code loading apparently becomes the choice at the cost of raising privacy and security concerns to mobile users.

Third, we may also consider ways to design ad libraries that satisfy the needs of advertisers, ad networks and users alike [21, 22, 32]. As in traditional web-based ad libraries, these systems display targeted advertising and report the network impressions, click-throughs, etc. to bill the advertiser. However, they aim to do these things irrefutably yet anonymously. The ultimate aim is to only provide the ad network with the metrics needed for billing, while allowing the user to retain complete and direct ownership of

[2]While we only studied one particular platform, due to the similar nature of integrating in-app ads into smartphone apps, we expect similar privacy and security risks will also exist on other platforms.

personally-identifiable information. Unfortunately, each approach proposed so far has required either additional overhead (extra data transfers, extra storage on the device, etc.), an organizational shift (third-party ad "dealers," the direct involvement of wireless providers, etc.), or both. As some ad libraries may not brand the ads that they serve, the user is usually ignorant of the ad networks used by an app. Therefore, these disadvantages may not be offset by competitive advantages for ad networks that operate in a privacy-preserving manner.

From another perspective, our current study is limited to those ad libraries that are simply "piggybacked" into host apps. Particularly, current ad libraries are typically self-contained (as a standalone package) so that they can be readily included by app developers. However, it is possible to have more advanced mechanisms (e.g., collusion [26], re-delegation [18, 20], or indirect channels [31]) that could avoid using dangerous Android APIs being modeled by AdRisk while still accessing various personal information on the phone. Note that there are some ongoing research projects that aim to detect or mitigate these attacks [9, 10, 11, 18]. How to extend AdRisk to seamlessly integrate these systems remains an interesting task for future work.

6. RELATED WORK

Smartphone privacy and security has recently attracted considerable attention. Researchers have employed various techniques to understand or assess these risks. For example, PiOS [12] used program slicing to detect privacy leaks in iOS apps. SCanDroid [19] analyzed Android apps' source code, along with the manifest file included with each app, to produce a data-flow policy specification that describes an app's use of information. Woodpecker [20] uses interprocedural data-flow analysis to detect possible confused-deputy attacks [23] on Android firmware. However, none of them is designed to understand or assess the information leaks and security risks from the embedded ad libraries. In contrast, AdRisk focuses on the risks from these ad libraries in the context of privacy (e.g., information harvesting) and security (e.g., untrusted code downloading and execution). In the case of SCanDroid, its reliance on the Java source code for an app renders it unable to analyze most ad libraries, which typically are only distributed in a compiled form.

TaintDroid [13] takes a different tack to expose and identify privacy leaks in apps as a whole. By using lightweight dynamic taint analysis built into modified Android middleware, the system alerts the user to the presence and nature of the leak. Note it is only concerned about the whole apps, not explicitly the ad libraries they contain. Also, as a dynamic technique, it may be able to precisely pinpoint possible leaks, but it is generally incomplete in not exploring all possible execution paths. Most recently, Enck *et al.* [14] wrote the ded Dalvik decompiler to study around one thousand popular Android apps, and reported a number of findings about them. In this work, we studied one hundred thousand apps, which allowed us to systematically identify and assess a wider variety of ad libraries. For example, none of the libraries that feature dynamic code loading (Section 4.2.3) were included or reported earlier. We believe that such dynamic code loading is dangerous, especially in light of recent Felt *et al.*'s findings [17], which are related to identifying "overprivilege" in Android apps. (An overprivilege occurs when an app requests more permissions than it uses.) In particular, among 940 Android apps being studied, more than one third were found to be overprivileged. Given the permission-probing behavior in existing ad libraries, it is possible that even more apps are requesting unnecessary permissions, which are then opportunistically being used by their embedded ad libraries. Dynamic code loading paints a yet more grim picture, as we found one ad library uploaded the permissions its host app was granted before downloading the code.

On the defensive side, several related solutions have been proposed and many of them revolve around the permission system. For instance, Kirin [15] checks the manifest of apps that are being installed against a permission-assignment policy, blocking any that request certain potentially unsafe combinations. Saint [28] takes this approach a step further, by allowing app developers to constrain permission assignment at install-time and permission use at run-time. Other systems try to add further expressivity to the permission system, such as Apex [27], which modifies the framework to allow permissions to be selectively granted and revoked at runtime. MockDroid [7] rewrites privacy-sensitive API calls to simulate their failure. TISSA [33] similarly protects user information, but instead does so by modifying the Android framework to support user-defined information disclosure policies; sensitive APIs can return false information under such a scheme instead of simply failing. AppFence [24] further refines this approach by adding taint-tracking, allowing yet more nuanced policies. However, these systems typically treat the apps as a while, without further differentiating the embedded ad libraries from hosting apps.

More generally, researchers have explored ways to deliver targeted ad content without disclosing any private information to the advertiser or ad network. For example, Adnostic [32] addresses the online ads and allows for behavioral web advertising without giving behavioral information to the ad network (by using a dedicated Firefox browser extension to prevent unnecessary information disclosure). MobiAd [22] takes a similar approach by using a broadcast mode available to wireless providers to stream a large amount of tagged ad content that is then filtered by mobile devices. Privad [21] offloads ad selection to the client, but aims to do so in a way that is less disruptive to the existing industry model for ad networks; in particular, much emphasis (including a follow-on work [30]) is placed on preserving the auction mechanism by which advertisers compete for ad slots on the networks. These systems incorporate cryptographic billing to ensure that click-throughs are properly billed to the advertiser without compromising the consumer's privacy, and without encouraging click fraud. Lastly, there are some efforts that specifically aim to address the privacy concerns inherent with location information. PrivStats [29] offers a mechanism so that aggregate location information can be irrefutably collected in a privacy-preserving way. Bindschaedler *et al.* [8] attempts to prevent tracking individual devices' movements by changing their identifiers in crowded regions. While these systems are making progress in mitigating the privacy risks, it is unclear yet whether they can be applied in our context to handle the in-app ad security risks (Section 4.2.3).

7. CONCLUSIONS

In this paper, we systematically examine the security and privacy issues raised by in-app ad libraries. We analyze 100 ad libraries selected from a sample of 100,000 apps collected from the official Android Market, and find that even among some of the most widely-deployed ad libraries, there exist threats to security and privacy. Such threats range from collecting unnecessarily intrusive user information to allowing third-party code of unknown provenance to execute within the hosting app. Since Android's permissions model cannot distinguish between actions performed by an ad library and those performed by its hosting app, the current Android system provides little indication of the existence of these threats within any given app, which necessitates a change in the way existing ad libraries can be integrated into host apps.

8. REFERENCES

[1] Android Permission Protection Levels. http://developer.android.com/reference/android/R.styleable.html#AndroidManifestPermission_protectionLevel.

[2] Android Security and Permissions. http://developer.android.com/guide/topics/security/security.html.

[3] Baksmali: A Disassembler for Android's Dex Format. http://code.google.com/p/smali/.

[4] Dalvik. http://sites.google.com/site/io/dalvik-vm-internals/.

[5] Distmo Report: April, 2011 and May, 2011. http://www.distimo.com/publications.

[6] Gartner Says Sales of Mobile Devices in Second Quarter of 2011 Grew 16.5 Percent Year-on-Year. http://www.gartner.com/it/page.jsp?id=1764714.

[7] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*, HotMobile '11, May 2011.

[8] L. Bindschaedler, M. Jadliwala, I. Bilogrevic, I. Aad, P. Ginzboorg, V. Niemi, and J.-P. Hubaux. Track Me If You Can: On the Effectiveness of Context-based Identifier Changes in Deployed Mobile Networks. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '12, February 2012.

[9] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, and B. Shastry. Towards Taming Privilege-Escalation Attacks on Android. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '12, February 2012.

[10] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry. Practical and Lightweight Domain Isolation on Android. In *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, CCS-SPSM'11, 2011.

[11] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[12] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS '11, February 2011.

[13] 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 Symposium on Operating Systems Design and Implementation*, OSDI '10, pages 1–6, February 2010.

[14] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[15] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 235–245, October 2009.

[16] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demysti.ed. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, October 2011.

[17] 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 (CCS '11)*, October 2011.

[18] A. P. Felt, H. Wang, A. Moschuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Security Symposium*, August 2011.

[19] A. P. Fuchs, A. Chaudhuri, and J. S. Foster. SCanDroid: Automated Security Certification of Android Applications. http://www.cs.umd.edu/~avik/papers/scandroidascaa.pdf.

[20] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '12, February 2012.

[21] S. Guha, B. Cheng, and P. Francis. Privad: Practical Privacy in Online Advertising. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI '11, March 2011.

[22] H. Haddadi, P. Hui, and I. Brown. MobiAd: Private and Scalable Mobile Advertising. In *Proceedings of the 5th ACM International Workshop on Mobility in the Evolving Internet Architecture*, MobiArch '10, pages 33–38, September 2010.

[23] N. Hardy. The Confused Deputy, or Why Capabilities Might Have Been Invented. In *ACM Operating Systems Review*, volume 22, pages 36–38, 1988.

[24] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. "These Aren't the Droids You're Looking For": Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*, October 2011.

[25] IDC. Android Rises, Symbian 3 and Windows Phone 7 Launch as Worldwide Smartphone Shipments Increase 87.2% Year Over Year. http://www.idc.com/about/viewpressrelease.jsp?containerId=prUS22689111.

[26] C. Marforio, F. Aurélien, and S. Čapkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical Report 724, ETH Zurich, April 2011.

[27] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 328–332, April 2010.

[28] M. Ongtang, S. E. McLaughlin, W. Enck, and P. D. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference*, ACSAC '09, pages 340–349, December 2009.

[29] R. A. Popa, A. J. Blumberg, H. Balakrishnan, and F. H. Li. Privacy and Accountability for Location-Based Aggregate Statistics. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*, October 2011.

[30] A. Reznichenko, S. Guha, and P. Francis. Auctions in Do-Not-Track Compliant Internet Advertising. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS '11)*, October 2011.

[31] R. Schlegel, K. Zhang, X. Zhou, M. Intwala, A. Kapadia, and X. Wang. Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, NDSS '11, pages 17–33, February 2011.

[32] V. Toubiana, H. Nissenbaum, A. Narayanan, S. Barocas, and D. Boneh. Adnostic: Privacy Preserving Targeted Advertising. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, NDSS '10, February 2010.

[33] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST '11, June 2011.

APPENDIX

	Included in Apps	Probes Permissions	Uses Obfuscation	Uses Reflection	Uses JavaScript	Read Installed Packages	Location Data	Place Phone Call	Camera	List Accounts	Read Calendar	Read Contact/Call Logs	Read Browser Bookmarks	Read Phone Information	Read Phone Number	Read SMS	Send SMS	Change Calendar	Change Contacts	Use Vibrator	ClassLoader
mopub/mobileads	87	✓	·	✓	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·
adfonic	77	✓	✓	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
vdroid	72	✓	✓	·	·	·	✓	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
transpera	72	✓	·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
mobgold	65	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
mobus	64	·	·	·	·	·	·	·	·	·	·	·	·	✓	✓	✓	·	·	·	·	·
hutuchong	63	✓	·	✓	·	·	·	·	·	·	·	·	·	✓	·	·	✓	·	·	·	·
mads	56	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·
everbadge	55	·	·	·	✓	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
zetacube/libzc	54	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
livepoint/smartad/sdk	52	✓	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·
l/adlib_android	45	✓	✓	·	·	·	✓	✓	·	·	·	·	·	✓	·	·	✓	·	·	·	·
eng/trickersticks	45	·	·	·	·	·	·	·	·	✓	·	✓	·	✓	✓	·	✓	·	·	·	·
ccmedia	36	✓	·	·	·	·	✓	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
sosceo	31	·	✓	·	·	·	✓	✓	·	·	·	✓	·	✓	✓	·	✓	·	✓	·	·
kr/netsco/Mojiva	29	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
admogo	28	✓	·	✓	·	·	✓	·	·	·	·	·	·	✓	✓	·	·	·	·	·	·
nexage/android	25	✓	·	✓	·	·	✓	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
rhythmnewmedia	23	✓	·	·	✓	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·
mediba/ad/sdk	22	✓	·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
madvertise	22	✓	✓	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·
qriously	20	✓	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·
zumobi/adslib	18	·	·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
netmite	15	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
imapp/ads	15	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
cn/yicha/android/ads	14	✓	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·
oneriot	13	·	✓	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·
aduru	11	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
admoda	11	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
jp/co/imobile	10	✓	✓	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
moblico	8	✓	·	✓	·	·	✓	·	·	·	·	✓	✓	✓	✓	·	✓	·	✓	✓	·
jp/ne/linkshare/android/tgad	8	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
ignitevision	8	✓	✓	·	·	·	✓	·	·	·	·	✓	·	·	·	·	·	·	·	·	·
fractalist	8	·	·	·	·	·	·	·	·	·	·	·	·	✓	✓	·	·	·	·	·	·
plankton	7	✓	✓	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	✓
cellitads	7	·	·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
cn/appmedia/ad	6	·	✓	·	·	✓	✓	·	·	·	·	·	·	✓	·	·	✓	·	·	·	·
glam/AndroidSDK	5	·	·	·	✓	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·
adtutu	5	·	✓	·	·	·	·	·	·	·	·	·	·	✓	✓	·	·	·	·	·	·
ru/adfox	4	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
netad	4	✓	✓	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·
freewheel	4	✓	·	✓	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	✓
adinside/androidsdk	4	·	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·
taobao/ads	3	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·	·
amaze/ad	2	·	·	·	·	·	·	✓	·	·	·	·	·	✓	·	·	·	·	·	·	·
admozi	2	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·
wqmobile/sdk	1	✓	·	✓	·	·	✓	✓	·	·	·	·	·	✓	✓	·	✓	·	·	·	·
ubermind/ad	1	·	·	·	·	·	·	·	·	·	·	·	·	✓	·	·	·	·	·	·	·
innoace/imad	1	✓	·	✓	✓	·	✓	✓	·	·	·	·	·	✓	✓	·	·	·	·	·	✓
AdyxSdk	1	·	·	·	·	·	✓	·	·	·	·	·	·	·	·	·	·	·	·	·	·
Total	1239	24	14	9	5	1	22	7	0	1	0	4	1	25	9	1	7	0	2	1	3

Table 4: The Overall Results from the Remaining 50 Ad Libraries