Efficient Privilege De-Escalation for Ad Libraries in Mobile Apps

Bin Liu*, Bin Liu†, Hongxia Jin*, Ramesh Govindan‡
*Samsung Research America, Mountain View, CA
†School of Computer Science, Carnegie Mellon University, Pittsburgh, PA
‡Department of Computer Science, University of Southern California, Los Angeles, CA
{bin2.liu, hongxia.jin}@samsung.com bliu1@cs.cmu.edu ramesh@usc.edu

ABSTRACT

The proliferation of mobile apps is due in part to the advertising ecosystem which enables developers to earn revenue while providing free apps. Ad-supported apps can be developed rapidly with the availability of ad libraries. However, today's ad libraries essentially have access to the same resources as the parent app, and this has caused significant privacy concerns. In this paper, we explore efficient methods to de-escalate privileges for ad libraries where the resource access privileges for ad libraries can be different from that of the app logic. Our system, PEDAL, contains a novel machine classifier for detecting ad libraries even in the presence of obfuscated code, and techniques for automatically instrumenting bytecode to effect privilege de-escalation even in the presence of privilege inheritance. We evaluate PEDAL on a large set of apps from the Google Play store and demonstrate that it has a 98% accuracy in detecting ad libraries and imposes less than 1% runtime overhead on apps.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, packages*; K.4.1 [**Computers and Society**]: Public Policy Issues—*Privacy*

General Terms

Design, Experimentation, Languages, Measurement, Security

Keywords

Previlege De-Escalation; Ad Libraries; Mobile Apps; Static Analysis; App Instrumentation

1. INTRODUCTION

Mobile device usage has reached astronomical levels in recent years. According to a recent report from Gartner Forecasts [12], Android device shipments alone are predicted to hit 1.17 billion by the end of 2014. Much of this growth has been fueled by two factors: the ease of app development, and the availability of free apps.

MobiSys'15, May 19–22, 2015, Florence, Italy.

http://dx.doi.org/10.1145/2742647.2742668.

The ease of app development has been enabled in part by system support for component-oriented development [10], in which app developers can easily integrate libraries (sometimes from third-parties) as components of their app. In the Android ecosystem, libraries are used for various purposes. Social SDKs, like Facebook, Twitter4j and WeChat, allow developers to integrate popular social elements to their apps. Development tools, such as ActionbarSherlock, HoloEverywhere and GeocoderPlus, provide developers with feature-rich and relatively well-maintained plugins for enhancing, say the utility, UI, and image processing capabilities of their apps.

The availability of free apps has been enabled by an ad ecosystem in which app publishers incorporate ad delivery software in their apps. This software is provided by ad networks who contract with advertisers to deliver ads to end users. App developers are paid by ad networks either by the number of times ads are seen by users, or the number of times they are clicked, or some combination thereof. More than 83% of apps [11] in Google Play are free and developers of free apps mostly rely on advertising for revenue [58].

To facilitate ad delivery, ad networks provide *ad libraries* to app developers. Ad libraries, like AdMob, InMobi and Vungle, offer developers solutions for monetizing their apps by showing ads to app users. To show ads in their apps, developers need to bundle ad SDKs in their app code to display ads to users through corresponding ad widgets. These ad widgets can communicate with their ad networks to fetch and show proper ads, according to apps' context or users' information, such as location.

In this ecosystem, incentives are skewed against the user. Developers have the incentive to deliver the most relevant apps to maximize their revenue, as do ad networks. This results in ad libraries taking unwarranted liberties with personal data on devices in order to more efficiently target ads. As a result, although users seem to understand the privacy risks with using mobile apps in general [19], they are especially concerned about privacy risks posed by ad libraries [54]. An extensive user study [43] points out that "mobile advertising services were a consistent privacy concern for the most participants" and users "felt the least comfortable when private resources were used for advertising." In another previous study [41] on location privacy of mobile advertising, users stated that they would "feel more comfortable" with advanced privacy controls than "with a simple opt-in/out-out mechanism".

One approach to this problem is to selectively *de-escalate privileges* to each component of an app. In this way, an ad library can have fewer privileges than the app logic itself. The Android access control model, which governs access to sensitive data such as location, identifiers, and contacts, and which has not evolved significantly over many releases, is also too coarse-grained to permit such privilege de-escalation. The current access control model allows users to specify *permissions* at app install time, and these permis-

sions apply to all components of the app. In response to this, there is a large literature that has explored finer-grained access control methods (Section 8). In general, these methods are effective but require significant changes to the underlying OS, or, when they do not require such changes, can sometimes be ineffective in enforcing access to sensitive data. Moreover, simple solutions that deny privileges to ad libraries that are not needed by the app logic are likely to be ineffective: our measurements show that in a majority of apps in a large corpus of over 60,000 apps, the permissions requested by the ad are a subset of the permissions needed by the app logic (Section 5).

In this paper, we propose a pragmatic approach to selective privilege de-escalation for ad libraries. In our approach, a user can allow the app logic access to location and the contacts database, but, if she chooses, deny these privileges to the ad library. Our system, called PEDAL, has several desirable properties: it *does not require OS or VM modifications*, is *resistant to obfuscated code*, *prevents privileges inherited by the ad library* from app code, and *is highly efficient*.

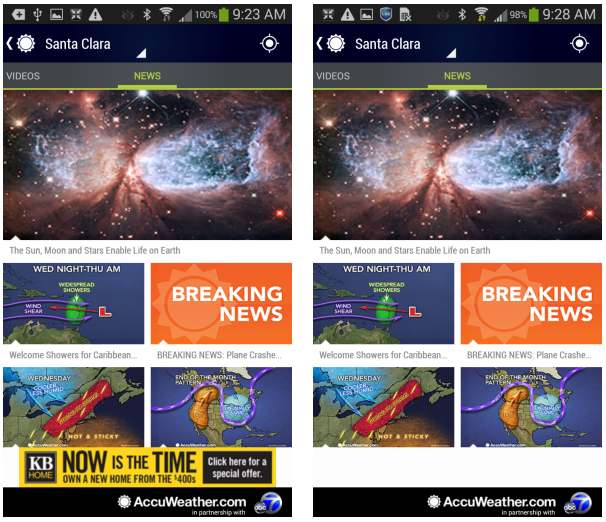
The design of PEDAL poses two challenges: how to identify ad library code in an app, and how to effect selective privilege de-escalation.

The first challenge is non-trivial because, in a compiled binary, there is no annotation that preserves the separation between bytecode from app logic and bytecode from an ad library. More important, library bytecode can often be obfuscated to prevent reverse engineering. In this paper, PEDAL surmounts these challenges by observing that, even in the presence of obfuscation, ad libraries have several features that can be used to identify them. These features arise from the functionality of ad libraries (ad delivery) and from their modularity (they have to expose well-defined interfaces to apps). Based on this observation, PEDAL includes a highly effective, obfuscation-resistant, machine classifier that can *separate* ad library from app logic code (Section 4).

Effecting privilege de-escalation is non-trivial also, since a pragmatic solution must not require changes to the OS or the VM, or must not require rooting[1] a phone, since these can adversely affect deployability. Moreover, any solution must be highly efficient; significant slowdowns in app execution time can affect usability. Most importantly, in a substantial fraction (25%, Section 6) of apps, ad libraries *inherit* privileges from the app logic — they access sensitive resources like location etc. by invoking callbacks to app code. Any solution for privilege de-escalation must prevent this kind of privilege inheritance. PEDAL uses efficient binary rewriting and information flow analysis to overcome these challenges (Section 4).

Figure 1 and Figure 2 show two examples of PEDAL in action. The first example shows how a user can, in the AccuWeather app, disable Internet access permissions for ad libraries but keep it enabled for this app's main logic. Figure 1 shows that while the app can still function (can download breaking news), the ad space is empty because the ad library cannot download ads. In the second example, in the AroundMe app, the ad library inherits location privileges from the app logic. In this case, the user can choose to use PEDAL's controller app, which allows users to specify de-escalation policies (see Figure 8 for details), to feed obscured location data. In the example shown, PEDAL feeds the correct location (San Jose) to the app, but feeds Sunnyvale (a city close to San Jose) to the ad library. Figure 2(b) shows the result: while the app lists hotels near the real location, the ad library recommends hotels around the obscured location.

[1]http://betanews.com/2013/10/01/5-reasons-not-to-root-android/



(a) Normal version (b) Controlled version

Figure 1: Disable Internet access for ad libraries.

(a) Normal version (b) Controlled version

Figure 2: Return obscured location for ad libraries.

PEDAL can be used both by end-users and app markets. Individual Android users may use this system to enable self-defined resource access control for ad libraries of apps on their phones. App markets may also apply PEDAL to enforce their policies on ad libraries. For example, an app market may provide *plausible-obscured-location* options to users to enjoy apps with obscured but plausible location targeted ads. For example, instead of feeding accurate location, say San Jose, to ad libraries, PEDAL may still feed plausible nearby locations, such as Santa Clara or Sunnyvale to them to require useful ads. In this scenario, the ad widgets cannot collect accurate location information but the ads can still be relevant to the user's current location.

As an aside, there is a fundamental difference between PEDAL and existing ad blockers such as AdBlock Plus [14]: Ad blockers attempt to completely prevent ad delivery, while PEDAL allows the option

of only blocking access of certain resource types to ad modules but still allowing ads to be delivered. In addition, ad blockers usually require rooted phones to enable full functionality while PEDAL does not.

Overall, this paper makes four contributions:

- The design and implementation of a system, PEDAL, for selective privilege de-escalation for ad libraries.
- An efficient, accurate, automatic and obfuscation-resistant method for identifying ad libraries embedded in app binaries. Our method uses a machine classifier based on features extracted using static analysis of bytecode.
- An efficient and flexible privilege de-escalation method that prevents privilege inheritance. This method uses binary rewriting, so does not require changing the Android OS or rooting the phone.
- Extensive experiments on a large set of real apps to show that PEDAL is both effective and efficient. Our prototype of PEDAL has 98.9% precision and 97.7% recall on identifying ad libraries and requires only 0.57 seconds per app for identification and rewriting. Apps incur less than 1% additional running time and can very effectively de-escalate privileges for ad libraries.

The rest of the paper is organized as follows: Section 2 provides background and motivates the design of PEDAL. Sections 3 and 4 discuss the architecture, and the design and implementation of PEDAL. Sections 5 and 6 evaluate the efficacy of ad library identification, and of privilege de-escalation, respectively. Section 7 discusses PEDAL usage scenarios and future refinements. Related work is described in Section 8, and the paper is summarized in Section 9.

2. BACKGROUND AND MOTIVATION

Before describing PEDAL, we provide some background on app execution, code obfuscation, and the app privilege model in Android.

2.1 Background

Libraries in Android. Android apps are written in a dialect of Java, and compiled down into a bytecode that is executed by the Dalvik Virtual Machine (DVM). Apps may include library code either for special purpose uses (such as image or video processing), or for analytics or ad delivery. In this paper, we focus on libraries for ad delivery: some apps can include more than one such library.

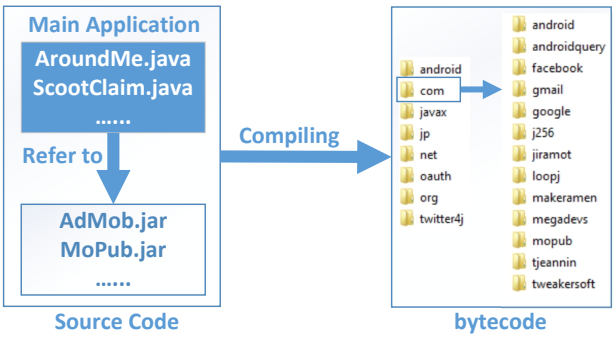


Figure 3: Source code to bytecode.

When developing an Android app, the developer sees clear boundaries between the app's source code and every other library. For example, as shown in Figure 3, the developers of the AroundMe app maintain their own source code. The app source also includes multiple libraries[2] provided by various third parties. However, when this app source is compiled to bytecode, the boundaries between the developers' code and libraries become unclear: compilation does not preserve all syntactic distinctions. Thus, it is sometime difficult to tell whether two classes belong to the app code or belong the a same library. One syntactic distinction that is preserved after compilation is the folder path for each class file. The folder path uniquely names a class file in a hierarchical namespace, and often, the upper levels of the hierarchy name the library that the class file belongs to.

Obfuscation. To prevent reverse engineering of source code from bytecode, app developers and library developers sometimes obfuscate the bytecode in two ways: at the package level and at the code-level. Package level obfuscation is motivated by prior work that has used folder path names to *statically* analyze app binaries to characterize potential risks [35], compute permission usage statistics [56], understand longitudinal code changes [26], extract usage patterns [57], and characterize apps [58] and so on. All of these pieces of work perform these analyses by matching bytecode paths against a manually-maintained blacklist of ad library names. For example, a path `/com/millennialmedia/android` indicates the presence of the ad library "Millennial Media". To foil this detection, some ad libraries, like AirPush, LeadBolt and AdsMogo, perform package name obfuscation and generate random package names for different developers.

Source code level obfuscation renames classes, methods and fields to semantically obscure names to prevent from reverse engineering the source code. Figure 4 shows a simple example of using package name obfuscation and source code obfuscation. Clearly, source code level obfuscation makes it harder to reverse engineer the original source from bytecode.

A nontrivial fraction of Android apps contain obfuscated code. We manually examined 200 randomly chosen apps from a corpus of about 60,000 (Section 5), and found that 107 of them contained ad libraries with source-code level obfuscation, and 37 of them obfuscated app code.

The Android Permissions Model. The Android permissions model governs app access to *resources*. A resource is an abstraction for data generated either by a hardware device (GPS or camera), data unique to a device (phone number, device ID, MAC address), or data generated by users and generally considered private (contact information). The Android permissions model requires an app developer 1) to notify users of which resources are required for the app, and 2) to explicitly ask the user's permission, at app install time, for access to these resources. Once the app is installed, the app and all its included libraries are granted access to these resources as long as the app is installed on the phone.

This model is at once too permissive and too coarse-grained, and there have been many examples of instances where permissions have been misused to breach privacy, often by ad libraries [26, 35, 55]. In response to this, recent versions of Android include a hidden permission manager called "App Ops" [1] that allows users to manage app permissions after installation. Moreover, several pieces of research have proposed finer-grained and restrictive access control for resources; we cover these in Section 8.

[2]Not included in this example are native libraries [6] that are written in native-code languages such as C/C++. The purpose of introducing native libraries is mainly for reusing existing code libraries written in C/C++. We discuss native libraries later.

```
package com.somecompany.ads.sdk
public class AdRequest {
    private final int requestId;
    private final String requestMessage;

    public AdRequest(int id,
                     String message) {
        requestId = id;
        requestMessage = message;
    }
    public int getId() {
        return requestId;
    }
    public String getMessage() {
        return message;
    }
}
```

(a) Source code

```
package com.ylzr.btpl187440
public class a {
    private final int a;
    private final String b;

    public a(int paramInt,
             String paramString) {
        this.a = paramInt;
        this.b = paramString;
    }
    public int a() {
        return this.a;
    }
    public String b() {
        return this.b;
    }
}
```

(b) Obfuscated code

Figure 4: Perform package name obfuscation and source code obfuscation on the source code

2.2 Privilege De-Escalation

Motivated by the shortcomings of the current permission model, we focus on a specific problem: how to selectively de-escalate the privileges given to ad libraries? In other words, even though a user may grant an app permissions to several resources, we would like to restrict access by ad libraries to some of these resources. Our position is that ad libraries may, in general need a different set of resources than the apps they are included in, so users should be able to selectively control privileges for these libraries separately from the main app logic.

Why focus on ad libraries? In Figure 5(a), a measurement of more than 60,000 apps shows that nearly 57% of apps use ad libraries. This indicates the magnitude of the problem: more than 1 of every 2 apps may be susceptible to privacy leakage. Furthermore, in roughly half of the ad-included apps, the ad libraries requested permission for a resource that was not required by the app logic itself!

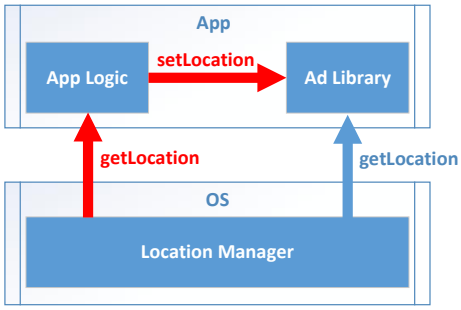


Figure 6: Two ways of resource access. Taking Location as an example, ad libraries can directly get the location information by calling system calls, or provide interfaces for the app logic to transfer the location information to it.

There are several ways in which to address this problem, but we consider a part of the design space that has not been explored before. We focus on a solution that has the following properties:

- It permits users to *selectively de-escalate* the privileges associated with ad libraries, without affecting privileges granted to other libraries or to the app itself.
- It is *obfuscation-resistant*: it has a high likelihood of success even in the presence of source-code obfuscation. This is motivated by the finding above that non-trivial amounts of obfuscation exist in today's apps.
- It must *not require changes to the operating system or the API*, or require special forms of modification (like rooting the phone). These assumptions enhance the deployability of the solution.
- It must be *efficient* and only minimally impact the execution time of the app.
- Finally, it must guard against *privilege inheritance* by ad libraries. In many apps, ad libraries, rather than access resources directly, invoke methods in the app logic that provide access to resources (Figure 6). Effectively, ad libraries inherit privileges granted to the app itself, and any privilege de-escalation must prevent this. In our corpus of 60,000 apps, we have found substantial numbers of apps that inherit access to location, the phone ID, and, to a lesser extent, the contact information and the account profile, as shown in Figure 5(b).

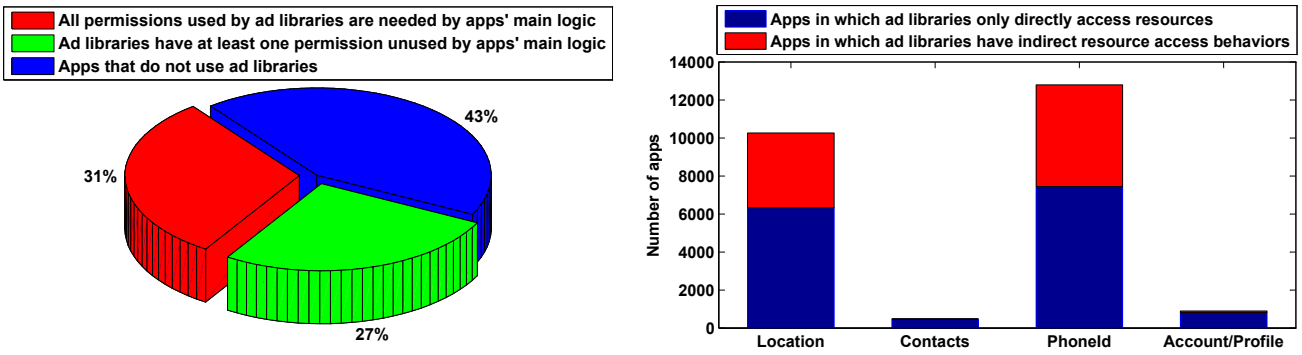
3. EFFICIENT PRIVILEGE DE-ESCALATION

In this paper, we describe the design and evaluation of a system for efficient privilege de-escalation. The system, named PEDAL, contains two parts: a Separator and a Rewriter (Figure 7). The input to PEDAL is a packaged app, and the output is a repacked app with de-escalated privileges for any (obfuscated) ad libraries in the app. In this section, we describe the overall architecture of PEDAL, leaving the detailed design and implementation to the next section.

The input app is first unpacked and transformed to Java bytecode, then sent to the Separator for feature extraction and classification. Feature extraction and classification is designed to separate the bytecodes to two different sets. The Separator scans and collects statistics on bytecodes[3] to generate feature vectors. Those feature vectors are input to a classifier which outputs two bytecode sets: the ad library set and the app logic set.

The Rewriter then performs binary rewriting on the bytecode sets. Specifically, the goal of the Rewriter is to interpose on 1) direct access to resources from both sets, and 2) calls from the app logic code to the ad library set that accesses resources (to prevent privilege inheritance, Section 2). Rewriter maintains a list denoting

[3]Note that we do not consider native code when performing classification because the usage of native code in ad libraries is minimal. (Section 6).



(a) Classify apps by ad libraries and permissions

(b) Direct and indirect resource access of ad libraries

Figure 5: A measurement of 63,105 apps

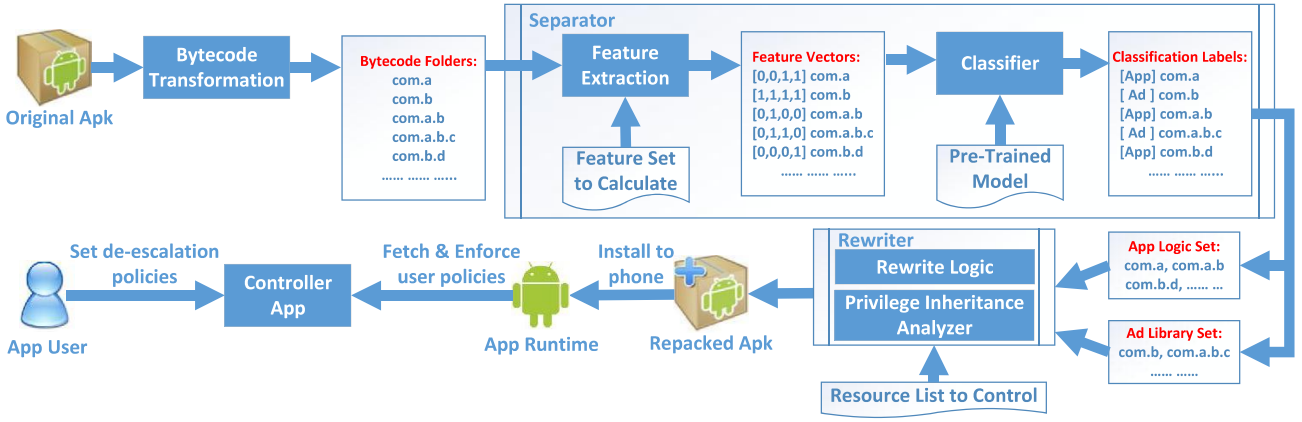


Figure 7: System Architecture

what private resources to control, such as Internet, location, contacts, phone ID and so on, takes both bytecode folder sets as inputs, and outputs the controllable versions (with respect to the resource types to control) of the two sets. As we describe later, the Rewriter does not attempt to interpose on all functions that lead to resource access, but 1) identifies a small set of *core resource access functions* and rewrites these functions which are used for *directly* accessing the concerned private resources, and 2) uses information flow analysis techniques to identify and rewrite another small set of functions which are used for ad libraries to *indirectly* access resources. The two rewritten bytecode sets, with other necessary files, are then repackaged back to a new app.

Finally, PEDAL contains a *Controller* app (Figure 7) for users to specify de-escalation policies for ad libraries.

This design achieves the requirements listed in Section 2 as follows. Classification and binary-rewriting achieve selective de-escalation on ad libraries. As we show later (and this is a major contribution of our work), the Separator is obfuscation-resistant. By using binary rewriting, our approach does not require OS level changes, and also achieves significant efficiency. Finally, the Rewriter, by analyzing information flow across bytecode sets, can prevent privilege inheritance.

4. PEDAL DESIGN AND IMPLEMENTATION

In this section, we present the detailed design and implementation of PEDAL.

4.1 The Separator

The goal of the Separator is to identify potentially obfuscated ad libraries packaged within an app. Before we explain the Separator, we observe that Java class files are usually named hierarchically. This naming hierarchy enables unique names for classes, permitting flexible inclusion of software from different sources into an application. Each bytecode corresponding to each Java class is then conceptually in a "folder" whose path name specifies the part of the hierarchy that class name is in. For example, the folder whose path is `/com/millennialmedia/android` would contain the ad library distributed by Millenial Media.

At a high-level, Separator marks each bytecode within an app as belonging to an ad library or not (i.e., it *separates* bytecodes, hence the name). Specifically, Separator identifies one or more folders that contain ad libraries. Folders can be nested, so a folder marked as containing an ad library implies that all bytecodes in subfolders are also part of the ad library.

As an aside, since Separator identifies ad libraries at the granularity of a folder, a potential approach to foil the Separator is to inter-mingle app logic and library code within a single folder. In practice, this is hard to do: the hierarchical class namespace, such as /com/google/android/gms/ads, /com/mopub/mobileads, /com/mobfox, /com/airpush/android, /com/mobclix/android/sdk, /com/applovin, and so on, is designed to enable separable development of code by different vendors, so that the folder hierarchy is used to distinguish code from different vendors. In this way, a developer can easily

incorporate class files from different vendors. Even within a given vendor, hierarchical naming is used to ensure modularity: for example, Google uses different hierarchical names for its analytics library and its ad library. Finally, we emphasize that while Separator relies on class hierarchies, it is robust to folder path name obfuscation, as described below.

The key idea behind Separator is to use a binary machine classifier trained to identify ad libraries, even in the presence of obfuscation. The classifier accepts a feature vector of a bytecode folder (described below) and predicts whether the folder belongs to an ad library. We have manually extracted labeled training data from several hundred ad libraries and apps to train the classifier, and the details are presented in Section 5. The main challenge in our work is selecting features for the classifier that would enable accurate classification in the presence of obfuscation.

Bytecode Extraction. Our approach does not require source code; rather it extracts semantic features from bytecode. For this, we convert Dalvik bytecode to Java bytecode using well known techniques [36], then perform feature extraction on the Java bytecode.

Feature Extraction. The most important aspect of Separator design is to choose the set of features that ensure high classification accuracy. The features need to be: 1) effective and accurate, correctly separating ad libraries from apps; 2) easy and fast, requiring few complex operations to calculate; 3) general enough to support a variety of ad libraries from different vendors; and 4) robust enough to support both obfuscated and non-obfuscated ad libraries. After examining and analyzing the bytecodes of popular ad library SDKs, we chose 128 features that meet the above goals and are informative to ad library classification. These features can be categorized into the following groups.

Usage of Android basic components: These are binary features of whether a bytecode folder contains the usage of Android basic components[4]: activity, service, content provider and broadcast receiver. Compared to other software libraries, ad libraries are usually implemented as an independent module. For example, they may contain activities for displaying ads, may hold background services for periodically requesting ads, may use content providers for storing and retrieving informational data, and may register broadcast receivers for better reacting to system events (e.g., disabled screen or low battery).

Usage of selective Android permissions: These are binary features of whether a bytecode folder uses the following Android permissions: *INTERNET*, *ACCESS_NETWORK_STATE*/*_WIFI_STATE*, *ACCESS_FINE*/*_COARSE_LOCATION*, and *WRITE_EXTERNAL_STORAGE*. These permissions are most likely to be used by ad libraries (they may, of course, be used by the app logic or by other components, so permissions usage is only one of the many features we consider). *INTERNET* is necessary for communicating with ad networks. *NETWORK_STATE* and *WIFI_STATE* are for checking the availability and status of network connections. *LOCATION* is for better ad targeting, and *WRITE_EXTERNAL_STORAGE*, which implicitly allows *READ_EXTERNAL_STORAGE*, is for saving and serving cached ads when the device is offline.

To determine this feature vector, we first extract API calls and content provider URIs from the bytecode, then map that information to permissions using PScout [21]. Note that we have also tried taking all Android permissions and treated each one as an individual feature. However, in practice, this causes model overfitting and degrades the classifier performance.

Usage of visual elements: These are binary features of whether a bytecode folder has implemented instances of visual elements. Ad libraries provide user interfaces (UI) for showing ads. For example, interstitial ads prefer to implement their own activities for displaying full-screen ads, and banner ads usually extend webview or imageview to show ads inside small rectangles. Features in this group check whether there are customized classes that are derived from the commonly used UI widgets, such as activity, webview and videoview. We identified 12 classes from *android.app*, *android.widget* or *android.webkit* that can be used for implementing ad UI components.

Usage of information sources and sinks: Information sources refer to system calls that read non-constant information from a shared resource, and information sinks refer to system calls that write a piece of information to a shared resource [51]. Ad libraries prefer certain types of information. For example, they often use account and phone state information to uniquely identify user devices. We extract features in this group by using SuSi [51], a machine-learning approach to classify Android system calls to 12 information source categories and 15 information sink categories. These source and sink categories form 27 features. Each binary feature indicates whether any API call belonging to a category is observed in a bytecode folder.

Usage of APIs for runtime permission check: One interesting observation, which was first pointed out in [56], is that sometimes ad libraries may try to dynamically check certain permissions and take advantage of these permissions if they are available to apps. This check can be performed through a cluster of API calls like android.content.Context.check*Permission (where "*" denotes a collection of API call names such as "checkCallingOrSelfPermission"). Thus, one feature is added to indicate whether a bytecode folder has called any functions for runtime permissions checks.

Keyword matching for class, method and field names: In practice, we have found that the combination of these features accurately recognizes most ad libraries. However, these features can sometimes wrongly classify social-network or e-business plugins into ad libraries, such as Facebook or Amazon. These plugins have very similar behaviors as ad libraries and do, indeed, have many of these features. Therefore, the added binary features in this group show, in a bytecode folder, whether the names of classes, methods or fields contain specific keywords that are often used in ad libraries but rarely used in social-network and e-business plugins, such as "ads," "adsdk," "interstitial," "campaign," "impression," "mraid (Mobile Rich Media Ad Interface Definitions)," and so on. We use a dictionary of 26 such keywords, and 78 (26×3) features for keywords defined for classes, methods and fields, respectively.

Obfuscation-Resistance. The features we have chosen for classification are robust to code obfuscation. First, code obfuscation is used to protect developer code from reverse engineering, and it does not obscure API calls, content provider URI or class names of the Android system. For example, ProGuard [15], an official Android obfuscation tool, explicitly confirms this in its manual. Obviously, except for the last group of features (keyword matches), all the features are only related to Android system entities and will not be affected by code obfuscation. Second, code obfuscation can foil keyword matching by obscuring most names in an ad library; however, it still needs to preserve some interface names so that can be easily understood by developers and simplify integration with applications. For instance, Chartboost [9], a popular ad library that uses obfuscation, still preserves some methods and classes with explicit and in-

[4]We refer interested readers to [5] for more Android application fundamentals.

formative names, such as *showInterstitial*, *requestMorePromotion* and *ImpressionActivity*, for ease of integration. Thus, in most cases, we can still capture features for the last feature group even with an obfuscated ad library. Actually, as shown in the evaluation, even without keyword matching features, the classifier can still make reasonable decisions based on other feature groups.

Finally, our features *will not be affected by package name obfuscation* because the Separator does not rely on the path or package name or components thereof, only on the class hierarchy itself. For example, if `/com/millenialmedia/android` is obfuscated to `/a/b/e`, Separator's features do not change and it will be able to identify all bytecodes in folder `/a/b/e` as belonging to an ad library. Path name obfuscation makes it harder to associate the ad library with its developer, but that association is not necessary for PEDAL's functionality.

An Optimization. Each Android app has a main bytecode package that usually carries the implementation of the most important part of the app logic. The name of the bytecode folder that contains this is globally unique across an app store, and is defined in the app's manifest. Instead of attempting to classify this main bytecode package, we directly mark this folder as *not* being an ad library. This optimization has two advantages: 1) it can prevent the potential influence of wrongly classifying this major functional folder as an ad library; and 2) it can speed up the total classification time by skipping feature vector generation for these folders – classification time matters because it can impact how quickly an app store can process a batch of apps.

Augmenting Separator with call-graph information. The classifier itself focuses on finding the key functional components of ad libraries such as components for fetching/displaying ads or sending user data to ad networks, so it may miss some auxiliary libraries referenced by the key functional components. These libraries are still logically part of the ad library, but may not contain any of the signatures of these libraries. We applied call-graph information to attempt to ascertain these auxiliary libraries. However, as we show in Section 5, doing this incurred significant cost without additional benefit: intuitively, if these auxiliary libraries do not have signatures similar to ad libraries and perform some auxiliary function (e.g., parsing JSON objects), it is probably not necessary to instrument them to perform privilege de-escalation. So our PEDAL prototype does not use this technique.

4.2 The Rewriter

The output of the Separator is input to the Rewriter, which effects efficient privilege de-escalation by binary re-writing based on user-specified privacy policies. We begin by first describing user-specified policies permitted in PEDAL, then describe the Rewriter design.

User-Specified Policies. In PEDAL, users specify policies using a Controller app (Figure 8). This controller app is accessed through a unique URI, and a special provider permission is set up to prevent unauthorized apps from accessing it. This provider permission is granted to every rewritten app by adding a permission claim entry to their AndroidManifest.xml files using the AXML library [8]. At runtime, a rewritten app can access the content provider by using *queryController* (details are explained below), to read the user's configuration and effect user-specified policies. Conceptually, each policy entry is a quadruple:

(*AppName*, *ResourceType*, *ComponentType*, *Policy*)

where *AppName* is an app's identifier, *ResourceType* is one of the five resources listed in Table 1, *ComponentType* takes the values {*AD*, *APP*}, and dictates whether the policy is specified for

```
// the original function is:
// Location LocationManager.getLastKnownLocation(
//              String provider)
public static Location PEDAL.getLastKnownLocation(
          LocationManager manager, String provider) {
    Policy currentPolicy = queryController("AccuWeather",
              "Location", "AD");
    if (currentPolicy == Policy.BLOCK) {
        return null;
    }
    else if (currentPolicy == Policy.OBSCURE) {
        Location obscuredLocation = new Location(provider);
        // set obscuredLocation to Sunnyvale
        obscuredLocation.setLatitude(37.369);
        obscuredLocation.setLongitude(-122.036);
        return obscuredLocation
    }
    // queryController returns Policy.ALLOW
    return manager.getLastKnownLocation(provider);
}

private static Policy PEDAL.queryController(appName,
          resourceType, componentType) {
    // query user's configuration
    // resourceType is:
    // Internet, Location, a content provider URI, etc.
    // componentType is: "AD" or "APP"
}
```

Figure 9: Example of rewriting a normal system call (This figure and Figure 10 use Java-style pseudo-code to describe the instrumentation technique for ease of understanding. PEDAL rewrites byte-code, and does not require access to Java or Dalvik source.)

the ad library or the app logic, and finally, *Policy* takes the value from {*ALLOW*, *OBSCURE*, *BLOCK*}. Our current instantiation of PEDAL supports the five resource types listed in Table 1, and users may thus selectively permit or deny an ad library access to each of these resource types. These five resource categories cover the types of resources accessed by ad libraries. It is easy to extend PEDAL to support other resource types.

In addition, users may choose to obscure the returned results for specific resource types. App developers may derive revenue from some of these resources. For example, when an ad library can obtain a user's location and deliver targeted ads, the developer might get more revenue than when no location is available. So, PEDAL permits users to specify policies that enable some form of obscured result to be returned. We discuss this below in some detail.

Finally, users can change policy settings at any time, and policy changes take immediate effect (even while an app is running). The Rewriter accesses these policies at run-time through a *queryController* interface, as discussed below.

Rewriting for Privilege De-Escalation. Conceptually, re-writing for privilege de-escalation is straightforward: Rewriter can interpose on resource accesses by the ad library or the app logic, and permit access only if the user-specified policies grant access. In the example in Figure 9, the call to *queryController* permits access to the Location resource if the policy permits, otherwise it returns null or an obscured location. However, there are several subtleties in the design, as described below.

Interposing on content providers. Rewriting content provider[5] based resource access is slightly more involved. Figure 10 shows another

[5]Content providers are basic Android components, and are actually databases addressable by their application-defined URIs. One use of content providers is to share information between apps.

Figure 8: Controller App Design

Table 1: Core Functions for Different Resource Types

Resource Type	Related Permissions	Number of Core Functions	Core Function Examples
Internet	INTERNET ACCCESS_NETWORK_STATE	15	java.net.URLConnection.openConnection org.apache.http.HttpResponse.execute
Location	ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION	12	android.location.LocationListener.onLocationChanged android.location.LocationManager.getLastKnownLocation
Contacts	READ_CONTACTS	8	android.content.ContentProvider.query android.content.CursorLoader.<init>
Phone ID	READ_PHONE_STATE ACCESS_WIFI_STATE	6	android.telephony.TelephonyManager.getDeviceId android.net.wifi.WifiInfo.getMacAddress
Account/Profile	GET_ACCOUNTS READ_PROFILE	5	android.accounts.AccountManager.getAccounts android.accounts.AccountManager.getAccountsByType

```
// the original function is:
// Cursor ContentResolver.query(Uri uri,
//            String[] projection, String selection,
//            String[] selectionArgs, String sortOrder)
public static Cursor PEDAL.query(
           ContentResolver resolver, Uri uri,
           String[] projectin, String selection,
           String[] selectionArgs, String sortOrder){
    // uri.toString() is something like:
    // content://com.android.contacts, content://call_log
    Policy currentPolicy = queryController("Facebook",
              uri.toString(), "APP");
    if (currentPolicy == Policy.BLOCK) {
        return null;
    }
    else if (currentPolicy == Policy.OBSCURE) {
        // Uri for obscured content provider
        Uri obscuredUri = PEDAL.getObscuredUri(uri);
        return resolver.query(obscuredUri,
                  projection, selection,
                  selectionArg, sortOrder);
    }
    // queryController returns Policy.ALLOW
    return resolver.query(uri, projection,
              selection, selectionArg, sortOrder);
}
```

Figure 10: Example of rewriting a system call for operating a content provider

example for rewriting the API call *query*. This function can be used to access all types of resources that are implemented through content providers, and the *uri* parameter is used to differentiate which resource type is actually queried. For example, if *uri* is "content://com.android.contacts" or "content://contacts," then the resource type is contacts; if it is "content://call_log," then the resource type is call logs; and if it is "content://com.android.calendar," then the resource type is calendar. When *queryController* receives a content provider URI as the *resourceType* parameter, it first maps the URI (using PScout [21]) to the resource type the URI pertains to, and then follows the same procedure as when it directly receives a resource type string, like "Location" above.

Obscuring Results. Our current prototype permits users to obscure contacts, location, phone ID, and profile information. Specifically, for location, phone ID, and profile information, PEDAL initiates a new object with pre-defined fake values, and returns this new instance to the app logic or the ad library. However, as discussed above, access to contacts is through a special URI, and the return is a cursor pointer to the actual database. Therefore, it is a little tricky to *OBSCURE* contacts. We have implemented a fake content provider (essentially, an app) that contains all the fields that have been defined in the real contacts content provider tables. To *OBSCURE* contacts, PEDAL simply replaces the real provider URI with the URI of the fake content provider, and returns the usable cursor pointing to the fake database. We have left to future work to determine how best to obscure results that preserve user privacy without affecting the app developer's revenue stream.

Optimizing the Rewriter. The API for accessing a resource, like the Internet or Location, may have many calls, not all of which need to be interposed. For example, there are a total of 95 calls [4] for the Internet resource, but only about 15 of them (called *core resource access functions*, or core functions for short) need to be interposed on to enforce policy. The other calls provide either higher-level abstractions that ultimately call one of these core functions, or provide utility functionality (for example, for parsing headers, etc.). Avoiding interposing on the non core functions can reduce overhead (Section 5) by avoiding calls to *queryController* for dynamic policy checking. Table 1 lists the number of core functions for each of the 5 resource types we support.

Preventing Privilege Inheritance. As we have discussed, ad libraries may also access resources indirectly through the app logic. To prevent this, we use an information flow analysis tool, FlowDroid [20] with a reported accuracy of over 90%. Our usage of flow analysis is constrained: we are interested in code paths from calls to resource access core functions in the app logic to Internet access calls in the ad library. Such code paths indicate the potential for privilege inheritance misuse. Note that our approach only focuses on

```
// the original sharing function is:
// void AdRequest.setLocation(Location location)
public static void PEDAL.setLocation(
           AdRequest ad, Location location) {
    Policy currentPolicy = queryController("AroundMe",
              "Location", "AD");
    if (currentPolicy == Policy.BLOCK) {
        // do nothing
        return;
    }
    else if (currentPolicy == Policy.OBSCURE) {
        // set obscuredLocation to Sunnyvale
                .....
        ad.setLocation(obscuredLocation);
        return;
    }
    // queryController returns Policy.ALLOW
    ad.setLocation(location)
}
```

Figure 11: Example of rewriting a resource sharing function for location sharing to an ad library.

those kinds of privilege inheritance that can lead to misuse (actual leakage of data).

Once these potential leakage paths have been identified, Rewriter performs the same kind of interposition as above. Consider the "Location" resource. For this, we first find the core resource access functions that the app logic uses to fetch location information (information flow sources), and then find the places within the ad library part where the Internet core functions are called (information flow sinks). Then, we apply FlowDroid to search for all paths between the defined flow sources and sinks, and further find and interpose on all unique sharing functions that cross the boundary between the app logic and the ad library, as shown in Figure 11. However, in some cases, we cannot fake a resource type in a sharing function if the resource value has been transformed somewhere along the path. For example, it is possible that, a piece of "Location" information, say latitude 40.714 and longitude -74.006, is accessed in the app logic, and then transformed into a string "New York", and finally what is returned to the ad library part is the string instead of the actual location object. In these cases, because we do not have enough knowledge to analyze the semantics of the transfer between the original object type and the transferred object type, we conservatively block or return a null object for these sharing functions regardless of users' policy setting.

Other Details. Java reflection can foil our interposition technique, since reflection can invoke any method of an object at runtime without explicitly exposing the actual function signatures. To deal with this, PEDAL intercepts function calls to *reflect.Method.invoke* to decide the actual function signatures during runtime using the function call *reflect.Method.toGenericString*, and then distributes the *invoke* call to the actual control logic if core functions are dynamically detected. A similar technique can be used if reflection is used to achieve privilege inheritance. However, PEDAL cannot handle native libraries; in Section 5, we show that ad libraries rarely use native code.

After replacing all core functions and sharing functions with their rewritten versions, the modified bytecode together with other app files are then repacked to a modified package file. The repacking tool we have implemented strictly preserves and follows the procedure and parameters of the original packing process. Finally, the app is signed; we use the strategy introduced in [62] to re-sign apps using a parallel set of randomly generated keys.

Table 2: Feature Selection and Classifier Performance

Feature Selection	Category	Recall	Precision
All Features	Ad Library	0.991	0.988
	Non-Ad Module	0.976	0.982
	Weighted Average	0.984	0.985
No Keyword-Matching Features	Ad Library	0.857	0.866
	Non-Ad Module	0.827	0.815
	Weighted Average	0.842	0.841
Only Keyword-Matching Features	Ad Library	0.746	0.716
	Non-Ad Module	0.925	0.919
	Weighted Average	0.836	0.818
No Basic-Component Features	Ad Library	0.908	0.905
	Non-Ad Module	0.913	0.916
	Weighted Average	0.911	0.911
No Permission Features	Ad Library	0.899	0.899
	Non-Ad Module	0.887	0.902
	Weighted Average	0.893	0.901
No Visual-Element Features	Ad Library	0.937	0.928
	Non-Ad Module	0.928	0.925
	Weighted Average	0.933	0.927
No Sources-and-Sinks Features	Ad Library	0.922	0.91
	Non-Ad Module	0.902	0.884
	Weighted Average	0.912	0.897
No Permission-Check Feature	Ad Library	0.97	0.973
	Non-Ad Module	0.955	0.949
	Weighted Average	0.963	0.961

5. EVALUATION: THE SEPARATOR

In this section, we systematically evaluate the PEDAL classifier. The highlights of our evaluation are:

- PEDAL is highly accurate (98.9% precision, 97.7% recall) in separating ad libraries from apps. The system is robust to obfuscation: it can identify obfuscated ad libraries with 97.4% accuracy.
- PEDAL can discover nearly 5× more ad library sources than the number of ad libraries in blacklists prepared by prior work [26, 27, 35, 44, 49, 55, 57, 58].

The Dataset. We crawled the Google Play Store [13] and downloaded 63,105 free apps between June 2nd, 2014 and August 1st, 2014. About 5% of these apps failed the Dalvik to Java bytecode conversion step, so our analysis uses the remaining 59,759 apps.

Classifier Training. We use LibSVM [29], a state-of-the-art SVM library, to build our classifier. We obtained training samples of the classifier, which are ad and non-ad modules, from two sources. First, we collected 100 JAR files of popular ad SDKs listed in [7,35]. From the JAR files we extracted 335 ad modules as positive training data points, among which 117 use code obfuscation and 26 use package name obfuscation. Second, we collected the top 500 free apps in Google Play on August 1st, 2014, and extracted non-ad bytecode modules from these APK files. We randomly sampled 335 non-ad modules among them as negative training data points. The kernel (sigmoid) and parameters (γ=0.125, d=3, $cost$=8) of the SVM model were chosen according to a grid search of parameters in exponential scale. After a 10-fold cross validation, our classifier performed with accuracy of 98.5% on classifying ad and non-ad modules. To demonstrate that our classifier is not biased by including popular examples in the training set, we intentionally omitted Google AdMob, the most popular ad library, from the training set.

Feature Efficacy. We now quantify the accuracy loss when omitting each feature selectively (Table 2).

First, keyword-matching features which represent the occurrences of key phrases, such as "ads," "promotion," in class, function, or

field names, play an important role in boosting the accuracy of our classifier. If we exclude these features, the classifier would perform with an accuracy of 84%, which is lower than using the full feature set. Ad libraries appear to preserve such human readable names even in obfuscated library code, since this simplifies integrating the library into the app.

We then explored classifier accuracy if the classifier were to use only keyword-matching features. Compared to only using the remaining feature set (i.e., without keyword matching), these two approaches appear to have comparable accuracy. However, the accuracy of detecting ad libraries using only keyword-matching is very low. Taking these two results together, our results show that using keyword matching is necessary, but not sufficient (in the sense that other features are also necessary), to ensure high-accuracy.

Finally, omitting other groups of features selectively resulted in accuracy losses of between 2%∼9%, suggesting that every feature group is important for the classification. As an aside, we have explored many other feature sets related to function argument types, function return types, class inheritance, full permission list and so on. These additional sets only marginally increase accuracy, and can sometimes reduce accuracy due to classifier overfitting. Thus, the finally chosen features reflect the best tradeoff between the classification accuracy and the complexity of the feature space.

Classifier Evaluation on Real-world Apps. To further evaluate our classifier on real-world usage scenarios, we tested the classifier on a subset of real apps. We randomly chose 200 apps from our corpus of free Android apps. For each app, we applied the classifier to automatically identify the ad libraries in it, and examined the results by manually checking the bytecode to see whether the classifier can accurately and completely find the ad libraries within the app. Specifically, we checked ad-related behaviors by scanning through string elements, data format for URL requests and sensitive API calls in the bytecode of these apps, since this information cannot be obfuscated. Of the 200 apps, the classifier accurately and completely identified ad libraries in 186 apps (93.0%). From the 200 apps, we found 357 ad library occurrence from 174 unique ad libraries, of which the classifier identified 347. Note that among these 357 ones, 173 of them (48.5%) did not appear in the classifier training set (AdMob takes the largest number of 78). Among these apps, we found obfuscated bytecode in 120 of them (107 of them have obfuscated codes in ad library modules). Our classifier can still automatically capture 97.4% of the ad library appearances in these obfuscated bytecode blocks. In total, considering the performance on each individual app, the average precision of the classifier is 98.9% (median=100%, sd=6.0%), and the average recall is 97.7% (median=100%, sd=9.7%).

New Ad Libraries. We then applied PEDAL to our entire collection of 59,759 apps. Among all these apps, our system found 34,167 apps (57.17%) that contain at least one ad library. Moreover, our classifier discovered many ad libraries previously unknown to us.

PEDAL discovered 2,598 unique ad library modules, such as the popular "com.millennialmedia.android," and "com.google.ads", as well as the unpopular "com.linxad" and "com.aarki." Note that this count is only based on folder names, and obfuscation might cause this estimate to be off. For example, two essentially different ad libraries can have the same folder name "com.a.b" after obfuscation. Conversely, a single ad library may be obfuscated differently in different apps.

Table 3: Top-10 Frequently Occurring Ad Libraries by Ad Network Names

Name of ad library source	Frequency in the 34,167-app corpus
Google AdMob	21,772
Millennial Media	3,871
Flurry	3,215
InMobi	1,929
MoPub	1,777
Mobclix	1,581
Airpush	1,316
MobFox	1,092
Leadbolt	926
Tapjoy	899

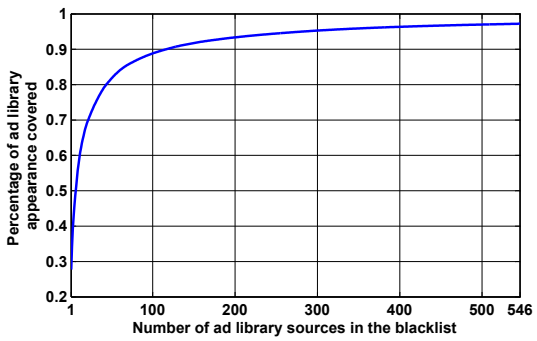
Among these 2,598 ad library modules, we identified 546 unique ad library sources[6] by: 1) retaining modules that are used by at least three different app publishers and 2) grouping modules that are known to be with a same ad network or have the same package prefix. The average occurrence of ad library sources (among the collection of 546 identified sources) in each app is 2.12 (max=35, min=1, median=1, sd=2.18). Table 3 presents the most frequently occurring ad libraries in our collection of apps with respect to the 546 unique sources.

Compared to the initial 100 ad library sources we applied in classifier training, our system discovered many more new ad libraries. This number is at least 5× more than the reported numbers in other static code analysis based papers that maintain blacklists of ad library package names [26, 27, 35, 44, 49, 55, 57, 58]. These findings suggest that a blacklist-based approach may be ineffective in identifying ad libraries. To quantify this, Figure 12 measures the (a) the percentage of apps that contain at least one library in the Top-*N* most popular ad sources (a single ad source may contain multiple ad libraries) and (b) the percentage of apps whose ad libraries are entirely contained in the Top-*N* most popular ad sources. When *N* is 100 (respectively, 200), these two values are 88.84% and 85.78% (respectively, 93.37% and 90.50%). Indeed, even if we used all the 546 ad library sources we discovered in a blacklist, these two percentages are 97.23% and 95.69%. This is because there are some highly obfuscated ad library package names that cannot be used in the blacklist: for example, an obfuscated package name "a.b.c" cannot be added to the blacklist because in different apps it may represent different functional modules after obfuscation.

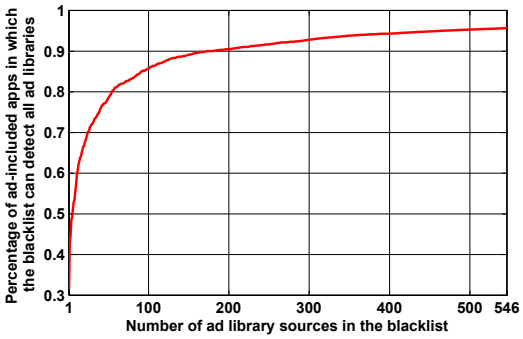
The Efficacy of Call Graph Information. As discussed in Section 4, our classifier is designed to capture core functional modules of ad libraries, and may miss some auxiliary modules. Therefore, we explored whether it would be beneficial to detect these auxiliary modules using call-graph analysis. To do this, we used Soot [17] to generate the class-level call graph of an app, then marked any auxiliary module as an ad library module if it was transitively called from a known ad library and was not a Android system library. We tested the efficacy of this approach on a random subset of 8,693 apps using ad libraries.

Call graph generation failed on 730 (8.4%) apps due to code obfuscation. For the remaining 7,963 apps, call graph analysis increases the classification cost by *over two orders of magnitude*, a factor of 143.12 on average (max=7,690.0, min=1.47, median=106, sd=347.5). On the other hand, this approach identified 684 new ad module occurrences (7.07% of all 9,676 occurrences), of which only 55 (0.8%) have requested new permissions, compared to ad libraries identified using the non-call-graph method. Thus, in general, this

[6]Modules that have obfuscated or randomized package names are excluded from grouping. Thus, the actual number of unique sources may be more than this number.



(a) Percentage of ad library occurrences covered



(b) Percentage of ad-included apps in which the blacklist can detect all ad libraries

Figure 12: CDF comparing with the blacklist approach

Table 4: Top-10 Frequently-Used Permissions

Used by apps' main logic (among 59,759 apps)		Used by Ad Libraries (among 546 Ad Sources)			
		Actually Used by Ads		Checked for Availability by Ads	
Permission	Frequency	Permission	Frequency	Permission	Frequency
INTERNET	43,498	INTERNET	546	ACCESS_FINE_LOCATION	104
ACCESS_NETWORK_STATE	36,485	ACCESS_NETWORK_STATE	413	ACCESS_COARSE_LOCATION	101
READ_PHONE_STATE	15,265	READ_PHONE_STATE	264	ACCESS_NETWORK_STATE	86
VIBRATE	14,299	RECEIVE_BOOT_COMPLETED	246	INTERNET	81
WAKE_LOCK	13,656	WAKE_LOCK	208	READ_PHONE_STATE	81
ACCESS_FINE_LOCATION	13,205	ACCESS_COARSE_LOCATION	190	WRITE_EXTERNAL_STORAGE	71
ACCESS_COARSE_LOCATION	11,153	ACCESS_FINE_LOCATION	173	ACCESS_WIFI_STATE	70
ACCESS_WIFI_STATE	6,987	ACCESS_WIFI_STATE	137	CALL_PHONE	39
RECEIVE_BOOT_COMPLETED	5,984	GET_ACCOUNTS	111	READ_CONTACTS	35
READ_CONTACTS	4,943	READ_CONTACTS	49	GET_ACCOUNTS	32

approach increases cost significantly while providing diminishing returns.

Is privilege de-escalation necessary? If the permissions used by ad libraries are disjoint from the ones used by the apps' main logic, we can simply apply tools such as App Ops [2] to disable those permissions requested by the ad library. Table 4 shows the statistics of the most frequently used permissions by both the ad libraries[7] and the app logic. There is significant overlap in the permissions requested by apps and by the ad library; of our nearly 60,000 apps, 29,926 apps have at least one permission common to both app logic and ad library. Of these, we found 15,988 apps (53.43%) in which the permissions used by ad libraries are *ALL* used by its app function. This motivates PEDAL's design, which enables finer-grained permissions than the underlying Android model.

PEDAL's design also foils ad libraries that access resources without explicitly requesting permissions. These libraries check available permissions at runtime, and sometimes use permissions granted to the app that includes the library. We analyzed both the bytecode and the documentation of 30 randomly selected ad library SDKs, and found that 5 of them have undocumented usage of important privacy-related permissions. For example, neither "com.appflood" nor "com.applovin" claims any dependency on permissions ACCESS_FINE_LOCATION or ACCESS_COARSE_LOCATION in their documentation. However, in the bytecode of these two libraries, we found that they both check and have API calls that rely on the location-related permissions. Similarly, "com.greystripe" claims to only use Internet-related permissions, but we found that this library checks and makes API calls to permissions CAMERA and READ_PROFILE.

[7]Note that each ad library source may have multiple versions, and we dated every versions of a same source using the heuristic introduced in [26], Therefore, the permission usage of an individual ad library source is based on its latest version found in our app corpus.

6. EVALUATION: THE REWRITER

In this section, we evaluate the overhead introduced by the Rewriter on 34,167 apps with at least one ad library. Our main results are:

- PEDAL processes an app in about a minute on a single thread. This task is trivially parallelizable.
- PEDAL introduces negligible runtime overhead (less than 1% on time cost), but can still effectively achieve privilege de-escalation.

The robustness of the Rewriter. The process of binary rewriting and re-packaging an app can sometimes be fragile: the resulting app may sometimes not install or run on a phone. To quantify this, we took the 34,167 apps that contained an ad library and re-packaged them. This step generated no failures. Then, we tested how many of these re-packaged apps would install correctly on an Android device (Galaxy S3). Of these, 32,965 (96.48%) apps installed perfectly; the ones that failed to install were due to dexopt failures. Finally, we used the PUMA [37] Android UI-automation framework to run each repacked app for 3 minutes or until all UI states were explored (whichever is shorter). In each app, the ad libraries' privilege was restricted: access to a resource either generated an exception or returned obscured or null data. In total, 10 Android emulators were created on a server running Ubuntu 14.04 with 20-core 2.8GHz CPU and 200GB memory and the corpus of 32,965 apps was equally partitioned across these emulators. Of these apps, 3,287 apps could not function properly and crashed mainly because of unhandled exceptions. For example, some apps call *HttpClient.execute* but fail to deal with the cases in which

Table 5: Micro-Benchmark: Running Time Cost of Each Step

Processing Step	Time Cost (in seconds, on 29,678 ad-included apks)				
	Average	Median	Max	Min	Stdev
Apk to Java Bytecode	5.81	4.52	52.24	0.48	5.01
Module Classification	0.57	0.24	11.95	0.00	0.81
Indirect Flow Analysis	28.93	8.58	455.44	0.34	49.99
Java Bytecode Rewriting	3.43	1.67	72.70	0.04	4.93
Repacking to Apk	17.82	10.54	617.86	1.99	27.77
Total time	56.63	25.98	1075.85	2.73	93.25

Table 6: Top-10 Frequency of Native Function Names

Used by apps' main logic		Used by Ad Libraries	
Function Name	App Freq	Function Name	App Freq
nativeInit	1266	UnitySendMessage	195
nativeRender	1115	getAsString	113
nativeResize	930	getAsBool	108
nativePause	905	getAsInt	108
nativeResume	872	newObject	98
nativeDone	852	getActivity	74
init	710	getLength	52
installNdk	609	getObjectAt	52
checkLibraryVersion	599	dispatchStatusEventAsync	46
newPolygonShape	571	getAsDouble	13

IOExceptions are thrown, although these exceptions are supposed to be properly handled according to API specs. In summary, PEDAL is reasonably robust end-to-end: out of the 34,167 apps, 29,678 (86.86%) were crash-free and successfully reported PEDAL debug tags. In the rest of this section, we report results on this set of apps.

Rewriting overhead. Table 5 shows statistics for all steps discussed in Section 4. Classification and rewriting incur, on average, 0.57s and 3.43s for each app respectively. Flow analysis for detecting privilege inheritance is as expensive as repacking in some cases when multimedia resources need to be re-packed. Overall, the mean per-app processing time is 56.62s, which is acceptable, especially for app stores, since apps can be processed in parallel.

Execution overhead. The runtime overhead of an app is the cost of invoking *queryController*. Our micro-benchmarks for this call revealed an average running time of 13.09 ms (sd=22.87 ms) for a normal resource, and 13.45 ms (sd=23.34 ms) for the URI resource. Other invocations (e.g., those dealing with invocations) cost less than 1ms.

To quantify the overhead end-to-end, we selected 100 apps which used the most number of ad libraries and also accessed the location resources. Executing these 100 apps on PUMA [37] showed a total increase in runtime of 0.89% (min=0.02%, max=1.81%, sd=0.51%) relative to unmodified versions of these apps[8]. This suggests that PEDAL introduces negligible overhead, so is eminently practical.

How often do ad libraries inherit privileges? With our experimental infrastructure, we are also able to quantify instances where ad libraries inherit privilege. We find that there are 7,499 apps out of the 29,678 (more than 25%) that access at least one of the five resource types through the app logic; this motivates Rewriter's careful information flow based analysis. Phone ID is the most commonly inherited resource, in 5,348 apps, and Location is next in 3,934 apps. Other resources are rarely inherited (about 0.5% in total).

Does PEDAL need to handle native code? We performed static analysis on the apps to search for calls on native code functions and found there are 5,381 apps (18.13%) that use native calls[9]. However, out of the 5,381 apps, only 314 apps (1.06% of the total 29,678 apps) use native calls in ad libraries. Table 6 lists the names of the top-10 used native function calls, generated by disassembling app binaries using `objdump`[10] and `IDA Pro`[11]. None of these appear to access resources. Finally, we did not find any instance of app logic invoking native code defined in ad libraries, suggesting privilege inheritance is unlikely. For this reason, PEDAL does not perform native code analysis.

How effective is PEDAL's privilege de-escalation? To quantify this, we ran 3 versions of the experiments on the 100 apps discussed above. Our UI automation tool replayed the same click stream to each version, and collected screenshots of each app page visited. In the first experiment, we analyzed the original apps without rewriting them. The 100 apps generated 1000 pages, and the unmodified app showed 843 ads. We manually checked and found that 304 of these ads were targeted at San Jose, the phone's location. In the second experiment, we disabled Internet access for the ad libraries. The apps after rewriting generated only 9 ads, indicating that selective network de-escalation was effective. (These 9 ads were generated by an ad library that used an HTTP client which invoked a core function for network access that was not in our set). In the third experiment, we obscured location data by feeding a fake location in New York, NY for a phone located in San Jose, CA. Among the 806 ads shown, 249 were New York based, while only 23 targeted the real location San Jose, CA. It is not trivial to determine how many ads should have been location targeted because the targeting strategies are hard to reverse engineer. However, in the unmodified app set, we observed that 304/843 or about 36% of the ads were targeted. In this set, if the same proportion were to hold, we would expect to see about 290 targeted ads, and our manual analysis uncovered about 272, which is close to the expected number. Thus, we feel confident that PEDAL's miss rate (the rate at which it fails to catch resource accesses) is relatively low (approximate 23/272). The 23 ads that targeted the San Jose location were attributed to two causes: the failure of the classifier to identify ad libraries, and the failure of the information flow analysis to detect privilege inheritance. Overall, these results suggest that obscuring location is generally effective; our corpus of 100 apps includes apps with the most ad library sources, so PEDAL's efficacy is likely to be higher for other apps than reported here.

PEDAL & ART. Android 4.4 introduced a new runtime, ART[12], which has officially replaced Dalvik in the newly released Android 5.0. Other proposed fine-grained control systems, such as the popular XPrivacy [18], cannot work with ART[13] because they are tightly coupled with the Dalvik environment or the old OS architecture. However, we have verified that PEDAL modified apps work with ART without any noticeable changes, since PEDAL does not require OS or runtime support.

7. DISCUSSION

Failures in Privilege De-Escalation. PEDAL's privilege de-escalation is not perfect: as shown in the evaluation, the Separator can introduce both false positives and false negatives, which can lead to wrongly classifying a social plugin as an ad library, or to an unstable app. However, these infrequent situations, users can choose to revoke and turn off privilege de-escalation for an app by using the *Controller* (Figure 8).

Another mechanism that may make the de-escalation more robust and efficient is that the app store can include developers in the loop, so that they can test the instrumented apps throughly and

[8]In running these experiments, we carefully ensured that both versions of the app were fed identical click streams by the "monkey".

[9]This number is slightly higher than the number of 17.41% reported by a recent large-scale study [58].

[10]http://en.wikipedia.org/wiki/Objdump

[11]https://www.hex-rays.com/products/ida/

[12]http://source.android.com/devices/tech/dalvik/art.html

[13]https://github.com/M66B/XPrivacy#FAQ75

understand the de-escalation effects before publishing the rewritten apps. However, developers may not always have the incentive or be willing to cooperate for fear of losing advertising revenue.

Resigning Apps. The last step of PEDAL is to resign a repacked app using a randomly generated developer key. This approach has two problems. First, automatic app updates can no longer be delivered because the signature of the app does not match the one from the original developer. However, in this case, the app store can actively monitor the app's update and prepare repacked versions for each update. Second, some features of the original app may break. For example, a repacked app may show a blank map if this app uses Google Maps API because the API certificate is tied to the original developer key used to sign the app which has been replaced during repacking. We have not found more effective solutions for this problem other than involving the developer.

Future Improvements. PEDAL can be improved along several dimensions:

Its classifier accuracy can be improved by designing better features or using more advanced classifiers. Specifically, the current selected features only reflect the most general characteristics of existing ad libraries, and we may need to re-design some of these features as ad libraries evolve. Moreover, although the designed features are resistant to package-level and code-level obfuscation, it is unclear whether they also work for other advanced obfuscation techniques, such as flow obfuscation/logic structural obfuscation, incremental obfuscation and so on. While we have not discovered uses of these advanced obfuscation techniques in today's ad libraries, in future work we plan to explore how effective our features are for such techniques.

In addition, although our core function list is fairly comprehensive, future work might wish to extend this set as the Android API evolves. Fortunately, PEDAL is designed to be extensible, and adding new core functions is simply a matter of adding some signatures. Improved information flow analysis tools can also improve the accuracy of PEDAL; for example, the tool we use, FlowDroid [20] has known limitations with respect to reflection.

Finally, PEDAL cannot detect the use of native code for resource access. Native code cannot bypass resource permissions since these are enforced by the kernel, but can bypass PEDAL's privilege de-escalation. As we show above, today's ad libraries do not appear to use native code for resource access. This is primarily because native code cannot access Android abstractions such as services and content providers [6]; these provide apps with access to location, the contact database, call logs, account/profile, etc. Also, recently, Android terms for ad networks enforce user tracking through a unique Google Advertising ID [3] assigned to each user, which also cannot be accessed by native code. Native code can use lower-level APIs for Internet access (sockets). In this case, PEDAL can effect privilege de-escalation by collaborating with a transparent Web proxy, such as Sandrop [16], an approach we have left to future work. Thus, most of the resources protected by PEDAL cannot be accessed natively, and for the Internet resource, a plausible solution exists. In the future, PEDAL can protect other resources (if any) that might be accessible by native code, by extending binary rewriting to native code, perhaps using Dynamic Binary Translation techniques [42, 50, 63], and we have left this to future work.

Extensions and Generalizations. PEDAL's contributions go beyond privilege de-escalation. Our classifier-based approach can be used to identify bytecode sets with different properties, such as social network or e-business plugins, or to train classifiers to recognize multiple categories. Moreover, PEDAL's privilege de-escalation can be combined with methods that implement other forms of fine-grained access control to support advanced privacy policies such as "allow location data access for ad libraries only during daytime," "block Internet access to http://www.hotads.com for ad libraries."

More generally, the strategy we have investigated in the paper is to first *separate and distribute code fragments to different functional sets*, and then *control each separated component individually*. This strategy should be generally applicable to other mobile platforms, though the details of the approach will differ for different platforms. For example, Windows Mobile platform uses C# (and C# bytecode), so it should be quite straightforward to extend our technique to support it by using bytecode analysis and rewriting frameworks for C#, such as CCI (Common Compiler Infrastructure)[14]. iOS uses Objective C, and other work has used binary rewriting for iOS apps as well [60], so we believe PEDAL's approach can be extended to iOS apps as well.

Reconciling Controllability and Usability. PEDAL can be treated as an extension to the default model of Android resource access control. However, while providing finer-grained options, it also increases the information workload for users to manage the settings, and, thus, may be preferred only by experienced or expert users. One way to address this extra workload is to use the technique of Privacy Revelations [61] to provide more details about how users' private data are spread among the app logic part and the ad library part so that app users can better understand and manage these settings. Another possible solution is to apply profile-based methods [44, 45] to reconcile privacy and usability to reduce users' burden of managing the settings by learning users' general preferences and providing smart default settings for users.

Possible Countermeasures. Ad libraries or app developers could foil PEDAL by: (1) Performing code obfuscation even for public APIs; in this case, developers may need to use some confusing APIs, such as "a.b" or "b.c.d" to integrate and call ad libraries, and this can reduce the accuracy of our Separator to around 84% according to Table 2; (2) Instead of using third-party ad libraries, developers can implement their own ad libraries (and may have to set up their own ad networks) and mix the implementation together with app logic, in which case, it could be very difficult to make any meaningful separation. While these countermeasures are feasible, they require significantly more development efforts.

8. RELATED WORK

Our work is inspired by two strands of work: one on selective privileges and fine-grained access control in mobile systems, and the other on analyzing malicious behaviors within the ad ecosystem.

Selective Privileges. Most closely related to PEDAL is prior work that has considered the precise form of selective privilege assignment for ad libraries. AdSplit [55] achieves this goal by extending the Android OS so that the application logic and the ad library run in different processes with different user IDs. AdDroid [49] instead suggests introducing a new set of API calls and permissions for ad libraries to Android system. LayerCake [53] modifies Android to permit ad libraries to embed UI elements in the main logic, without exposing data or privileges of the main app. Finally, Compac [59] modifies the Android OS to enable component level privileges, and developers need to manually distinguish components. Unlike these systems, PEDAL does not require modifications to the underlying operating system or its APIs, and does not require developer intervention.

Fine-grained Access Control. A complementary set of systems has considered fine-grained access control to resources. One line of

[14]https://ccimetadata.codeplex.com

work in this area has explored access control and information flow control techniques and associated expressive policy specification languages. Starting with TaintDroid [33], a long line of work has explored these techniques and languages: AppFence [38], MockDroid [25], Apex [48], TISSA [64], FlaskDroid [28], ASM [47] and ASF [22]. Another line of work has explored library re-writing [62] or loader modifications [18] to achieve the same goal. A third line of work has used binary rewriting (using techniques discussed in [31,36]) to achieve fine-grained access control [23,24,32,39,40,52] and derived and enforced sub-permissions in popular permission groups. None of these pieces of work attempt to enforce per-component access control; PEDAL is complementary and separates privileges per component, and PEDAL can be combined with some of these to provide finer-grained per component access control.

Ad Analysis. We have also been inspired by another line of work that explores the static or dynamic behavior of ad libraries. For example, several pieces of work have used static analysis to explore ad library security and privacy [35], the evolution of ad libraries [26], statistics of ad library usage on Google Play [58], characteristics of ad library APIs [27], users' common privacy preferences on ad libraries [44], and usage of ad-supported apps [57]. Our component separation technique builds upon the blacklisting techniques employed by these pieces of work: for example, [58] maintained a list of 21 ad libraries, [26] used a list of 68 ad libraries, and [35] generated a list of 100 ad libraries. In contrast, PEDAL is able to identify over 5× more ad library sources than prior work.

Finally, PEDAL is complementary to prior work that has used dynamic analysis to identify ad fraud [30, 46], or to estimate the impact of app plagiarism on ad revenue [34].

9. CONCLUSION

In this paper, we have described PEDAL, a system to achieve selective privilege de-escalation for ad libraries. PEDAL performs automated classification to identify ad library code, and rewrites core resource access functions and resource sharing functions of these bytecode sets to achieve de-escalation. PEDAL is robust, by design, to both package name obfuscations and source code obfuscation. Despite this, PEDAL shows remarkable classification accuracy and efficacy, yet requires reasonable computing power to process apps. Finally, PEDAL is effective and imposes negligible runtime overhead for apps.

Acknowledgements

We would like to thank our shepherd, Landon Cox, and the anonymous referees, for their insightful suggestions for improving the technical content and presentation of the paper.

10. REFERENCES

[1] Android 4.3 Includes Hidden App Permissions Manager That Could Bolster Privacy & Security. http://techcrunch.com/2013/07/26/android-app-ops/, 2013.

[2] How to Restore Access to App Ops in Android 4.4.2+. http://www.howtogeek.com/177915/how-to-restore-access-to-app-ops.....-in-android-4.4.2/, 2013.

[3] Advertising ID. https://developer.android.com/google/play-services/id.html, 2014.

[4] Android API Guides. https://developer.android.com/guide/index.html, 2014.

[5] Android Application Fundamentals. http://developer.android.com/guide/components/fundamentals.html, 2014.

[6] Android NDK. http://developer.android.com/tools/sdk/ndk/index.html, 2014.

[7] AppBrain Statistics on Android Libraries: Ad Networks. http://www.appbrain.com/stats/libraries/ad, 2014.

[8] AXML: Read Write Android Binary XML and resources.arsc Files. https://code.google.com/p/axml/, 2014.

[9] Chartboost. http://www.chartboost.com/, 2014.

[10] Component-based Software Engineering. http://en.wikipedia.org/wiki/Component-based_software_engineering, 2014.

[11] Distribution of Free vs. Paid Android Apps. http://www.appbrain.com/stats/free-and-paid-android-applications, 2014.

[12] Gartner Says Worldwide Traditional PC, Tablet, Ultramobile and Mobile Phone Shipments to Grow 4.2 Percent in 2014. http://www.gartner.com/newsroom/id/2791017, 2014.

[13] Google Play. https://play.google.com/store, 2014.

[14] Install Adblock Plus for Android. https://adblockplus.org/en/android-install, 2014.

[15] ProGuard. http://developer.android.com/tools/help/proguard.html, 2014.

[16] Sandrop: Secure Android Proxy. https://code.google.com/p/sandrop/, 2014.

[17] Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/, 2014.

[18] XPricacy. https://github.com/M66B/XPrivacy#xprivacy, 2014.

[19] P. Aditya, B. Bhattacharjee, P. Druschel, V. Erdélyi, and M. Lentz. Brave New World: Privacy Risks for Mobile Users. In *Proc. ACM SPME*, 2014.

[20] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. L. Traon, D. Octeau, and P. McDaniel. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proc. ACM PLDI*, 2014.

[21] K. Au, B. Zhou, J. Huang, and D. Lie. PScout: Analyzing the Android Permission Specification. In *Proc. ACM CCS*, 2012.

[22] M. Backes, S. Bugiel, S. Gerling, and P. Styp-Rekowsky. Android Security Framework: Enabling Generic and Extensible Access Control on Android. *arXiv preprint arXiv:1404.1395*, 2014.

[23] M. Backes, S. Gerling, C. Hammer, M. Maffei, and von P. Styp-Rekowsky. AppGuard: Enforcing User Requirements on Android Apps. In *Proc. ETAPS TACAS*. 2013.

[24] A. Bartel, J. Klein, M. Monperrus, K. Allix, and Y. Traon. Improving privacy on android smartphones through in-vivo bytecode instrumentation. *arXiv preprint arXiv:1208.4536*, 2012.

[25] A. R. Beresfordand, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *ACM HotMobile*, 2011.

[26] T. Book, A. Pridgen, and D. Wallach. Longitudinal Analysis of Android Ad Library Permissions. In *Proc. IEEE MoST*, 2013.

[27] T. Book and D. Wallach. A Case of Collusion: A Study of the Interface between Ad Libraries and Their Apps. In *Proc. ACM SPSM*, 2013.

[28] S. Bugiel, S. Heuser, and A. Sadeghi. Flexible and Fine-grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *Proc. USENIX security*, 2013.

[29] C. Chang and C. Lin. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[30] J. Crussell, R. Stevens, and H. Chen. MAdFraud: Investigating Ad Fraud in Android Applications. In *Proc. ACM MobiSys*, 2014.

[31] B. Davis and H. Chen. Retroskeleton: Retrofitting Android apps. In *Proc. ACM MobiSys*, 2013.

[32] B. Davis, B. Sanders, A. Khodaverdian, and H. Chen. I-ARM-Droid: A Rewriting Framework for In-app Reference Monitors for Android Applications. In *Proc. IEEE MoST*, 2012.

[33] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth. TaintDroid: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proc. OSDI*, 2010.

[34] C. Gibler, R. Stevens, J. Crussell, H. Chen, H. Zang, and H. Choi. AdRob: Examining the Landscape and Impact of Android Application Plagiarism. In *Proc. ACM MobiSys*, 2013.

[35] M. Grace, W. Zhou, X. Jiang, and A. Sadeghi. Unsafe Exposure Analysis of Mobile In-app Advertisements. In *Proc. ACM WiSec*, 2012.

[36] S. Hao, D. Li, W. Halfond, and R. Govindan. Sif: A selective instrumentation framework for mobile applications. In *Proc. ACM MobiSys*, 2013.

[37] S. Hao, B. Liu, S. Nath, W. Halfond, and R. Govindan. PUMA: Programmable UI-automation for Large Scale Dynamic Analysis of Mobile Apps. In *Proc. MobiSys*, 2014.

[38] 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 *Proc. ACM CCS*, 2011.

[39] J. Jeon, K. Micinski, and J. Vaughan. Application-Centric Security Policies on Unmodified Android. *University of Maryland Computer Science Department Technical Report*, 2011.

[40] J. Jeon, K. Micinski, J. Vaughan, A. Fogel, N. Reddy, J. Foster, and T. Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proc. SPSM*, 2012.

[41] P. G. Kelley, M. Benisch, L. F. Cranor, and N. Sadeh. When are users comfortable sharing locations with advertisers? In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '11, pages 2449–2452, New York, NY, USA, 2011. ACM.

[42] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proc. USENIX Security*, 2002.

[43] J. Lin, S. Amini, J. Hong, N. Sadeh, J. Lindqvist, and J. Zhang. Expectation and Purpose: Understanding Users' Mental Models of Mobile App Privacy through Crowdsourcing. In *Proc. ACM UbiComp*, 2012.

[44] J. Lin, B. Liu, J. Hong, and N. Sadeh. Modeling Users' Mobile App Privacy Preferences: Restoring Usability in a Sea of Permission Settings. In *Proc. SOUPS*, 2014.

[45] B. Liu, J. Lin, and N. Sadeh. Reconciling Mobile App Privacy and Usability on Smartphones: Could User Privacy Profiles Help? In *Proc. WWW*, 2014.

[46] B. Liu, S. Nath, R. Govindan, and J. Liu. DECAF: Detecting and Characterizing Ad Fraud in Mobile Apps. In *USENIX NSDI*, 2014.

[47] A. Nadkarni and W. Enck. ASM: A Programmable Interface for Extending Android Security. In *Proc. USENIX security*, 2014.

[48] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proc. ASIACCS*, 2010.

[49] P. Pearce, A. Felt, G. Nunez, and D. Wagner. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proc. ACM ASIACCS*, 2012.

[50] M. Probst. Dynamic Binary Translation. In *Proc. UKUUG Linux DeveloperâĂŹs Conference*, 2002.

[51] S. Rasthofer, S. Arzt, and E. Bodden. A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks. In *Proc. NDSS*, 2014.

[52] N. Reddy, J. Jeon, J. Vaughan, T. Millstein, and J. Foster. Application-Centric Security Policies on Unmodified Android. *UCLA Computer Science Department Technical Report*, 2011.

[53] F. Roesner and T. Kohno. Securing embedded user interfaces: Android and beyond. In *Proc. USENIX Security*, 2013.

[54] S. Rosen, Z. Qian, and Z. Mao. AppProfiler: a Flexible Method of Exposing Privacy-related Behavior in Android Applications to End Users. In *Proc. ACM CODASPY*, 2013.

[55] S. Shekhar, M. Dietz, and D. Wallach. AdSplit: Separating Smartphone Advertising from Applications. In *Proc. USENIX Security*, 2012.

[56] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating User Privacy in Android Ad Libraries. In *Proc. IEEE MoST*, 2012.

[57] A. Tongaonkar, S. Dai, A. Nucci, and D. Song. Understanding Mobile App Usage Patterns Using in-App Advertisements. In *Proc. Passive and Active Measurement*, 2013.

[58] N. Viennot, E. Garcia, and J. Nieh. A Measurement Study of Google Play. In *Proc. ACM SIGMETRICS*, 2014.

[59] Y. Wang, S. Hariharan, C. Zhao, J. Liu, and W. Du. Compac: Enforce Component-level Access Control in Android. In *ACM CODASPY*, 2014.

[60] T. Werthmann, R. Hund, L. Davi, A. Sadeghi, and T. Holz. PSiOS: Bring Your Own Privacy & Security to iOS Devices. In *ACM ASIACCS*, 2013.

[61] D. Wetherall, D. Choffnes, B. Greenstein, S. Han, P. Hornyack, J. Jung, S. Schechter, and X. Wang. Privacy Revelations for Web and Mobile Apps.

[62] R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proc. USENIX Security*, 2012.

[63] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proc. IEEE Security and Privacy*, 2009.

[64] Y. Zhou, X. Zhang, X. Jiang, and V. Freeh. Taming Information-stealing Smartphone Applications (on Android). In *Proc. TRUST*, 2011.