# Free for All! Assessing User Data Exposure to Advertising Libraries on Android

Soteris Demetriou, Whitney Merrill, Wei Yang, Aston Zhang and Carl A. Gunter
University of Illinois at Urbana-Champaign
{sdemetr2, wmerril2, weiyang3, lzhang74, cgunter}@illinois.edu

***Abstract*—Many studies focused on detecting and measuring the security and privacy risks associated with the integration of advertising libraries in mobile apps. These studies consistently demonstrate the abuses of existing ad libraries. However, to fully assess the risks of an app that uses an advertising library, we need to take into account not only the current behaviors but all of the allowed behaviors that could result in the compromise of user data confidentiality. Ad libraries on Android have potential for greater data collection through at least four major channels: using unprotected APIs to learn other apps' information on the phone (e.g., app names); using protected APIs via permissions inherited from the host app to access sensitive information (e.g. Google and Facebook account information, geo locations); gaining access to files which the host app stores in its own protection domain; and observing user inputs into the host app.**

**In this work, we systematically explore the potential reach of advertising libraries through these channels. We design a framework called Pluto that can be leveraged to analyze an app and discover whether it exposes targeted user data—such as contact information, interests, demographics, medical conditions and so on—-to an opportunistic ad library. We present a prototype implementation of Pluto, that embodies novel strategies for using natural language processing to illustrate what targeted data can potentially be learned from an ad network using files and user inputs. Pluto also leverages machine learning and data mining models to reveal what advertising networks can learn from the list of installed apps. We validate Pluto with a collection of apps for which we have determined ground truth about targeted data they may reveal, together with a data set derived from a survey we conducted that gives ground truth for targeted data and corresponding lists of installed apps for about 300 users. We use these to show that Pluto, and hence also opportunistic ad networks, can achieve 75% recall and 80% precision for selected targeted data coming from app files and inputs, and even better results for certain targeted data based on the list of installed apps. Pluto is the first tool that estimates the risk associated with integrating advertising in apps based on the four available channels and arbitrary sets of targeted data.**

## I. Introduction

Advertisers aim to generate conversions for their ad impressions. Advertising networks assist them in matching ads to users, to efficiently turn impressions into conversions. We call the information that achieves this *targeted data*. Android smart phones (which we call 'mobiles' from here on) contain rich information about users that enable advertising networks to gather targeted data. Moreover, there is considerable pressure on advertising networks to improve the number and quality of targeted data they are able to offer to advertisers. This raises many privacy concerns. Mobiles often contain sensitive information about user attributes which users might not comfortably share with advertising networks but could make valuable targeted data. This, in turn, led to a substantial line of research on privacy and advertising on mobiles in two general areas: (1) strategies for detection and prevention [53], [43], [17], [31], [33], [18], [50], [52], [7], [47], [48], [35], and (2) architectures and protocols that improve privacy protections [19], [36], [40], [29]. The first of these approaches primarily provides insights into the current practices of advertisers and advertising networks. The second examines a future in which a changed advertising platform provides better privacy. However, some of the studies show that the development and use of targeted data on mobiles is modest at present [47]. This is at least partially because most applications (which we call 'apps' from here on) do not pass along information about users to the advertising network—through its ad library embedded in the app—unless the advertising network requires them to do so [35]. This leave open an important question: *what if advertising networks took full advantage of the information-sharing characteristics of the current architecture?*

In particular, when one wants to assess the privacy risk associated with an asset, she needs to take into account not only past and current hazardous behaviors but all allowed actions that can result in potential privacy loss [27]. In the case of opportunistic advertising libraries, a privacy loss is possible if such libraries have the ability to access private user information without the user's consent. Current app privacy risk assessment techniques [26], [25], try to detect when sensitive data leaks from an app. To achieve that, they employ static or dynamic analysis of apps and/or libraries. However, applying this sort of assessment is constrained by the apparent practices of the advertising libraries. For example, every time an ad library is updated, or a new ad library appears, such analysis must be performed again. To make things worse, some ad libraries load code dynamically, [18] which allow them to indirectly update their logic without dependency on the frequency of their host app's updates. In this way, any analysis dependent on current library behaviors is unreliable as the analysis can not predict the behavior of updated code or dynamically downloaded/loaded code. Thus, to assess such risks, we need to have a systematic way to model the potential data exposure to ad libraries independent of current or apparent practices. A privacy risk


NDSS '16, 21-24 February 2016, San Diego, CA, USA

http://dx.doi.org/10.14722/ndss.2016.23082

assessment should consider what an adversary is allowed by the system to do instead of only what she is currently doing. Our work takes the first step in this direction by modelling the data collection capabilities of ad libraries on an Android platform.

We model opportunistic ad networks based on their abilities to access targeted data on an Android platform through at least four major attack channels: protected APIs by inheriting the permissions granted to their host apps; reading files generated at runtime by their host apps and stored in the host apps' protected storage; observing user input into their host apps; and finally unprotected APIs, such as the `PackageManager.getInstalledApplications()` that allow the ad library to access platform-wide information. We further categorize these attack channels into two classes, namely the `in-app` and `out-app` exploitation class. The `in-app` class contains attack channels that are dependent on the ad library's host app. The protected API's, app local files and user input are examples of such channels. The `out-app` class contains attack channels that are independent of the host app. The public API's are an example of this. In particular, Grace et. al. [18] identified that the list of installed applications on a user's device—which can be derived from a public API on Android—raises privacy concerns. In this work we systematically explore how this information can be exploited by an adversary in practice. We demonstrate and evaluate how well such APIs can result in an adversary learning a user's targeted data. Based on our data exposure modeling, we have designed and developed a framework called `Pluto`. Pluto aims to facilitate assessment of the privacy risk associated with embedding an untrusted library into an app. We show that Pluto is able to reveal the potential data exposure of a given app to its ad libraries through the considered attack channels. We believe that frameworks like Pluto will be extremely useful to app developers who want to assess their app's potential data exposure, markets aiming to better inform their users about the privacy risk associated with downloading a free app, and users themselves. In addition, we hope that this will spur similar academic attempts to capture the capabilities of ad libraries and serve as a baseline for comparison.

**Contributions**: We outline the paper's contributions below:

• *New understanding*: We perform manual and static analysis on real world apps across all Google Play categories to better understand what is currently available to ad libraries. We reveal an increasing trend in ad networks to collect the list of installed applications on Android. We show that numerous targeted data exist in app local files, in the form of user attributes or user interests that third-party libraries can readily access. These findings highlight that ad networks could become more aggressive in the future incentivized by the vast array of targeted data being made available to them.

• *New techniques*: We design novel natural language processing techniques to demonstrate that it is feasible for an opportunistic ad library to infer user information from structured files on Android. We present a new similarity metric called droidLESK and provide preliminary evidence that it can facilitate context disambiguation in Android. We further illustrate how Frequent Pattern Mining (FPM) can be leveraged to assess private data exposure of an app through the out-app channel.

• *Targeted data inference from the list of installed applications*: We are the first to systematically explore what an adversary can learn from the list of installed applications on a mobile device. We demonstrate and evaluate how reliably such unprotected APIs can be utilized by an adversary as side-channels to infer a user's attributes (e.g. gender, age, marital status).

• *Design, implementation and evaluation of a mobile library risk assessment framework*: We design the first of its kind ad library risk assessment framework. We implement a modular extensible prototype based on this design, for discovering targeted data exposure at scale in Android apps and evaluate its performance. The prototype of Pluto demonstrates that the techniques introduced in the paper achieve good results in practice and provide a baseline for comparison with future efforts. We plan to open source Pluto—after thoroughly testing it—and all our experimental results online [2].

This paper is organized as follows: Section II provides background information. Section III models the opportunistic ad library we consider. Sections IV and V describe our studies of the capabilities of ad networks through in-app and out-app exploitation channels respectively. Section VI describes our design and implementation of Pluto. In Section VII, we present our evaluation of Pluto. In Sections VIII, we discuss the utility and limitations of our approach. In Section IX, we discuss related work and conclude the paper in Section X.

## II. Background

**Mobile Advertising**: Mobile advertising is a common way for app developers to monetize their apps. Usually app developers offer their apps for free and include one or more advertising libraries which are assigned space to place ad content in the apps' user interface. Data brokers, entities that collect user data and maintain user profiles, incorporate ad libraries in mobile apps (or load code in iframes of online publishers' web sites). These libraries commonly collect targeted data consisting of user attributes and interests to build more complete user profiles. This allows the data brokers to sell information to advertisers. For example, advertisers develop campaigns to promote products at the behest of businesses. These advertisers will collaborate with data brokers who know how to reach specific groups of users. There are a number of types of data brokers with a distinction sometimes made between ad networks versus ad exchanges, but for this paper we will simply refer to them all as ad networks. The advertisers can request that their ads be displayed to a specific segment of the population. Payment systems vary, but common ones pay for the number of impressions shown to the desired segment or pay for the number of users that click on the impressions. Either way, the better the ad network is able to define the most promising segment and reach the most parties in this segment, the more money will be made by all of the parties in the ecosystem.

**Android Protection Mechanisms**: Each app on Android is assigned a unique static UID when it is installed. This allows the operating system to differentiate between apps during their lifetime on the device, so it can run them in distinct Linux processes when launched. In this way Android leverages the traditional Linux process isolation to ensure that one app cannot

access another app's resources.[1] However, when developers include an ad library, or any type of library for that matter, it is treated as part of the host app. The operating system will assign one `UID` for the app as a whole, even though the library and host app have different package names. Every time an app is launched, the OS will assign a process identifier (PID) to the app and associate that with the app's UID. Again this PID is shared between the host app and its libraries that run within the same Linux process. As a result, the host app and the library components will also share privileges and resources, both in terms of Linux discretionary access control (DAC) permissions and in terms of Android permissions granted. The former allows the library to access all the local files the host app is generating. The latter allows it to use the granted permissions (*e.g.,* ACCESS_COARSE_LOCATION) to access other resources on the device (such as GPS), that can expose user information (such as her location).

This multifaceted ecosystem, where there are strong incentives for more data collection by all stakeholders, needs to be better understood. Studying the current practices of ad libraries is an important place to start. Indeed our community already found that ad libraries collect some types of data for themselves even without the cooperation (or with the implicit consent) of the host app developer. Such behaviors have been observed in the wild since 2012 [18] and as a routine practice today [35] for certain types of information. Nonetheless, to fully assess the privacy risk associated with embedding a library into an app, we need to take into account not only past and current behaviors, but also all allowed events that can lead to breaches of users' data confidentiality. This work aims to take the first step into the direction of modeling ad libraries, not based on previous behaviors but based on their allowed actions on the Android platform. We show how this can be leveraged to design a tool that can assess the targeted data exposure to ad libraries.

**NLP Techniques**: The NLP community has developed different approaches to analyze unstructured data. For example, NLP is used to parse user reviews online or user voice commands to digital personal assistants. Work focused on extracting grammatical information to understand what the user is trying to convey. Part-of-speech Tagging (POS Tagging), is a typical technique to achieve that. It is used to determine for each word in a sentence whether it is a noun, adjective, verb, adverb, proposition, and other part of speach. A common problem in NLP arises when one needs to perform word sense disambiguation. That is, to derive a given a word's semantic meaning. This can be challenging as a word might have multiple meanings and complex relationships with other words. To this end, Wordnet [32], an English semantic dictionary has been proposed, where the community tried to capture most of senses, of most of the English words. Wordnet also provides relationships between words, such as whether two words are synonyms, or connected with `is-a` relationship and so on. In essence, Wordnet is a graph with words as nodes and relationships as edges. To assist in better capturing the relationships between words, the community has developed multiple similarity metrics which are different ways to parse the Wordnet graph. For example, the LCH [28] metric, uses the shortest paths between two words to determine how similar the words are. To accurately determine which of the multiple senses of the word is the most appropriate, one needs to carefully select the right similarity metric or design a new similarity metric, and design her system in a way that incorporates domain knowledge. These are challenges we had to overcome in our work to enable extraction of targeted data from local files. Furthermore, our target files do not contain real words that can be used in an actual conversation but rather variable names. We explain how we handle all these challenges in Section VI.

[1]This is with the exception of apps signed with the same developer key. In that case, the apps can indicate in their manifests that they should be assigned the same UID.

## III. Threat Model

A risk is the potential compromise of an *asset* as a result of an exploit of a *vulnerability* by a *threat*. In our case, the assets are user targeted data, the threat is an opportunistic ad library, and a vulnerability is what allows the ad library to access targeted data without the device user's consent or the consent of the library's host app. Here, we examine the capabilities of the ad libraries to collect such data on an Android platform.

Because libraries are compiled with their host apps, are in extend authorized to run as the same Linux process as their hosts on an Android OS. Thus the ad library code and the host app's code will share the same identifier as far as the system is concerned (both the static UID and the dynamic PID). In essence, this means that any given ad library runs with the same privileges as its host app. Consequently, the libraries inherit all the permissions granted by the user to the host app. There is no way for the user to distinguish whether that permission is used by her favorite app or the ad libraries embedded in the app. This permission inheritance empowers the ad libraries to make use of *permission-protected APIs* on the device. For example, if an app granted the `GET_ACCOUNTS` permission, its libraries can opportunistically use it to retrieve the user's registered accounts (e.g., the email used to login to Gmail, the email used to login to Facebook, the email used for Instagram, Twitter and so on).

Furthermore, during their lifetime on the device, apps create *local persistent files* where they store information necessary for their operations. These files are stored in app-specific directories isolated from other applications. This allows the apps to offer seamless personalized services to their users even when they are not connected to the Internet. In addition this practice enables the apps to avoid the latency of accessing their clouds, provided they have one. Android offers a convenient way through its `SharedPreferences` class to store and retrieve application and user specific data to an XML file in its UID-protected directory. In that directory, apps can also create their own files typically using standardized formats such as `XML`, `JSON`, or `SQLite`. In this way, they can utilize widely available libraries and Android APIs to swiftly and easily store and parse their data. The ad libraries, running as the same Linux user as their host apps, inherit both the Linux DAC privileges and the SE Android MAC capabilities of their host apps. This allows them to access the app's locally stored files as their hosts would. Consequently, the ad libraries could read the user data stored in those files. Consider, for example, the app *My Ovulation Calculator* which provides women a platform to track ovulation and plan pregnancy. This app, listed under the *MEDICAL* category on Google Play, has been installed 1,000,000–5,000,000 times. By parsing the

app's runtime generated local files, an ad library might learn whether its user suffers from headaches, whether she is currently pregnant, and, if so, the current trimester of her pregnancy. All these are targeted data which advertisers can monetize [42], making them a valuable addition to ad libraries.

Moreover, an aggressive ad library could utilize its vantage position to peak on *user input*. In particular, such a library could locate all the UI elements that correspond to targeted data related to user input [34], [23] and monitor them to capture the data as they become available. For example, by monitoring the user's input on `Text Me! Free Texting & Call`, a communication app with 10,000,000–50,000,000 downloads, an ad library would be able to capture the user's `gender`, `age` and `zip code`. Note that these data constitute the quasi identifiers [44] proven to be enough to uniquely identify a large percentage of registered voters in the US.

Nonetheless, an ad library can exploit both the inherited privileges of its host app and the position on a user's device. Irrespective of the host app, the ad libraries, can make use of *public APIs* to learn more about the user. Such APIs are considered harmless by the Android Open Source Project (AOSP) designers and are left unprotected. This means that the apps can use those APIs without the need to request permissions from either the system or the user. In this work, we found that by merely acquiring the list of installed applications through such APIs, one can learn targeted data such as a user's `marital status`, `age`, and `gender` among others.

To model these attack channels, we further categorize them them into two classes, namely the `in-app` and `out-app` exploitation class. The `in-app` class contains attack channels that are dependent on the ad library's host app. The protected API's, app local files and user input, are examples of such channels. The `out-app` class contains attack channels that are independent of the host app. The public API's are an example of this. Through the rest of this work, we assume that an ad library can gain access to targeted data through permission-protected APIs, runtime-generated app local files, user input, and unprotected APIs.

## IV. Data Exposure through In-App Channels

Ad libraries can leverage their position within their host apps to access exposed targeted data. Some targeted data are dependent on what the host apps themselves collect from the users. An ad library can access such data by parsing the files its host app created at runtime to store such information locally, that is in its own UID-protected storage. Furthermore, it can inherit the permissions granted to its host app and leverage that privilege to collect targeted data through permission-protected APIs. Finally, it can peek on what the host app user inputs to the app. In this section, we explore what an ad library can learn through these in-app attack channels. We elaborate on our methodology and provide insights from real world examples. To gain insight on what an ad library can learn, we perform manual inspection of some real-world free apps. This way we can validate our assumptions about data exposure through in-app attack channels and further create ground truth for test data that we can use to do evaluations of our framework in subsequent sections.

We first cherry-pick a few free apps we selected for purposes of illustration. We downloaded the target apps from Google Play and used Apktool to decompile them. We located the packages corresponding to the Google AdMob advertising network library and located an entry point that is called every time an ad is about to be loaded. We injected our attack logic there to demonstrate how the ad library can *examine local files*. In particular, our logic dumps the database and xml files that the app has created at runtime. We then compiled the app and ran it on a physical device by manually providing it with some input. Here are some examples of what such an aggressive ad library could learn in this position (or what AdMob is, in principle, able to learn now).

`I'm Pregnant` helps women track their pregnancy progress and experience. It has 1,000,000–5,000,000 installations and is ranked with 4.4 stars [2] on Google Play. Our code was able to read and extract the local files created by the host app. After manually reviewing the retrieved files, we found that the host app is storing the weight of the user, the height, current pregnancy month and day, symptoms such as headaches, backache and constipation. It also recorded events such as dates of intercourse (to establish the date of conception) and outcomes like miscarriage or date of birth.

`Diabetes Journal` helps users better manage their diabetes. It has 100,000–500,000 installations and ranked with 4.5 stars on Google Play. Our code was able to extract the local files generated by the app. Manually reviewing these files, we found that it exposes the user's birth date, gender, first-name and last name, weight and height, blood glucose levels, and workout activities.

`TalkLife` targets users that suffer from depression, self-harm, or suicidal thoughts. It has 10,000–50,000 installations on Google Play and ranked with 4.3 stars. In contrast with the other two apps above, TalkLife stores the user information in a user object which it serializes and then stores in a local file. In this case, some knowledge of the host app allows our code to deserialize the user object and get her email, date of birth, and first name. Deserializing the user object also provided our library the user password in plain text.

Thus, if an opportunistic advertising library is included in apps like these, then a careful manual review of the apps will reveal some pathways to targeted data. At this point it helps to have a little more terminology. Let us say that a *data point* is a category of targeted data point values. For example, gender is a data point, whereas knowing that Bob is a male is a data point value. What we would like to do, is examine a collection of apps to see what data points they expose to ad libraries.

To explore these ideas and their refinement we develop three datasets listed in the first three rows of Table I. For the first, we make a list of the 100 most popular free apps in each of the 27 categories on Google Play to get 2700 apps. After removing duplicate apps, we were left with 2535 unique apps. We call this the *Full Dataset, FD*. From these we randomly selected 300 apps for manual review. From these apps we removed the ones that crashed on our emulator or required the use of Google Play Services. We will refer to this as the *Level*

[2] Applications on Google Play are being ranked by users. A 5-star application is an application of the highest quality.

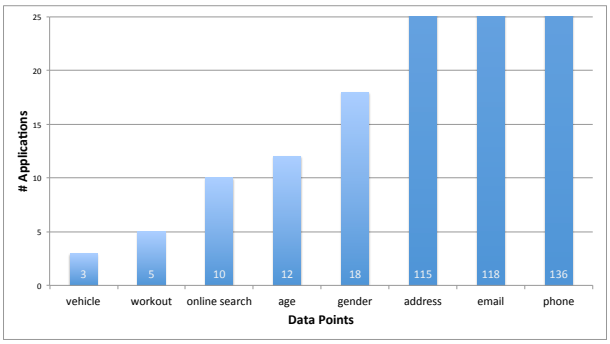
TABLE I: Datasets

| Name | Number | Description |
|---|---|---|
| Full Dataset (FD) | 2535 | Unique apps collected from the 27 Google Play categories. |
| Level One Dataset (L1) | 262 | Apps randomly selected from FD. |
| Level Two Dataset (L2) | 35 | Apps purposively selected from L1. |
| App Bundle Dataset (ABD) | 243 | App bundles collected through survey. |

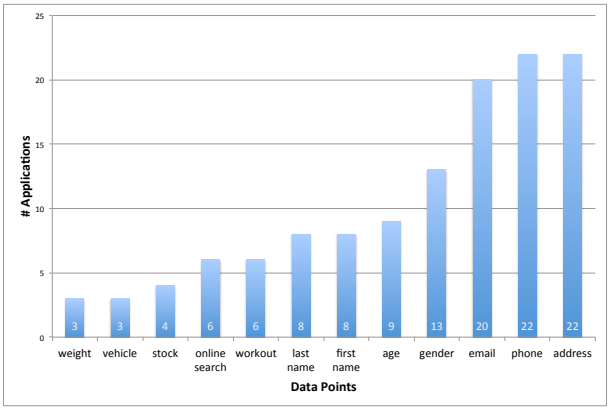
*One Dataset (L1)*. On this dataset, we searched for data point exposure by two means. First, we inspected the manifest to see if the permissions themselves would suggest that certain types of data points would be present. For example, we predicted that the address attribute could be derived by the library if the host app is granted the `ACCESS_COARSE_LOCATION` or the `ACCESS_FINE_LOCATION` permission, the email attribute from the `GET_ACCOUNTS` permissions, the phone attribute from the `READ_PHONE_STATE` permission and the online search from the `READ_HISTORY_BOOKMARKS` permission. Second, we launched the app, looked to see what local files it produced, and looked into these files to see if they expose any particular data points.

The data points we consider must include user data that the ad libraries are likely interested in harvesting. To this end, we extract data points mostly based on a calculator provided by the Financial Times (FT) [42]. This calculator provides illustrative information sought by data brokers together with an estimate of its financial value in the U.S. based on analysis of industry pricing data at the time the calculator was created. For example, according to the FT calculator, basic demographic information like age and gender are worth about $.007. If an opportunistic advertising network can learn that a user is (probably) an accountant, then the cumulative information is worth $.079 (according to the calculator); if they also know that this accountant is engaged to be married, this increases the value to $.179. Engaged individuals are valuable because they face a major life change, are likely to both spend more money and change their buying habits. An especially noteworthy data point is a pregnancy. This is well illustrated by events surrounding Target's successful program to use the habits of registered expecting shoppers to derive clues about unregistered ones in order to target them with advertising about baby care products [13]. The FT calculator provides us with a realistic way of exploring the relative value of an information gathering strategy. The precise figures are not important, and have probably changed significantly since the introduction of the calculator, but they give some ballpark idea of value and the system provides a benchmark for what a more accurate and detailed database of its kind might use.

We abstracted the questionnaire-like attributes from the FT calculator into keywords and used these as a guide to data points to find in the apps reviewed. For example, we transformed the question "Are you a fitness and exercise buff" into "workout". We refer to the overall attack technique that examines local files and uses protected APIs, as a *level one inspection* (L1-I). We found 29 categories of data points in L1 by this means, including 'gender', 'age', 'phone number', 'email address', 'home address', 'vehicle', 'online searches', interests like 'workout' and others. Table II depicts some popular apps and the data points they expose to ad libraries performing a *level one inspection*.




(a)




(b)

Fig. 1: Number of apps with data points inferred by (a) level one inspection of L1, (b) level two inspection of L2.

However, an ad library could also utilize the fact that it can *eavesdrop on user inputs* in its host app. This can be done on Android by exploring the resource files of packages. Once an interesting layout file is found, an offensive library can inflate the layout from the library package and read from its UI elements. With this strategy, the ad library can find targeted data that are input by the user but not necessarily kept in local files. Let us call the attack strategy that utilizes not only local files and protected APIs, but also user input eavesdropping, a *level two inspection* (L2-I). To better understand what data points are exposed to an ad library performing a level two inspection, we selected 35 of the apps in the L1 dataset and reviewed them manually to find data points that level two inspection could reveal. We call this the *L2 dataset*. The 35 apps in question are ones that exposed one or more data points other than ones derived from the manifest. We made this restriction to assure that there was no straight-forward strategy for finding data points in these apps so we could better test the automated inference techniques we introduce later. Table II depicts some popular apps and the data points they expose to ad libraries performing a *level two inspection*. We observe that apps expose not only demographic information but also more sensitive data such as user health information. The complete list of apps and the data points they expose is omitted due to space limitations.

Figure 1a displays the number of apks in the level one inspection that were found to expose the basic data points we listed earlier. Figure 1b portrays a similar graph for the level two inspection. Here, we pruned all data points with frequency less than three. We observe that data points that can be derived by exploiting the host app's permissions are more prevalent than other ones. This is because the permissions are coarse-

TABLE II: Data exposure from popular apps to ad libraries performing level-one (L1-I) and level-two (L2-I) inspection.

| Attack Strategy | Category | Application Name | Num. of installations | Exposed data points |
|---|---|---|---|---|
| L1-I | MEDICAL | Menstrual Calendar | $1 \times 10^6 - 5 \times 10^6$ | pregnancy, trimester, headache |
| L1-I | EDUCATION | myHomework Student Planner | $1 \times 10^6 - 5 \times 10^6$ | gender, age, address |
| L2-I | HEALTH & FITNESS | Run with Map My Run | $5 \times 10^6 - 10 \times 10^6$ | phone, email, first name, last name, age, gender, address, workout |
| L2-I | LIFESTYLE | BeNaughty - Online Dating App & Call | $5 \times 10^6 - 10 \times 10^6$ | phone, email, age, gender, address, marital status, parent |

grained and app developers are likely to use them for a number of reasons, whereas other data points would be present only if the host app is explicitly collecting that information. Overall, it is clear that targeted data is exposed by apps through in-app attack channels to ad libraries. Next we examine exposure through out-app channels.

## V. Data Exposure through Out-App Channels

Ad libraries can surreptitiously access targeted data not only through in-app attack channels but also from host-app-independent channels such as public APIs. Such APIs are considered to be harmless and thus made available to all applications on the platform without the need of special permissions. In particular, Android provides a pair of publicly available functions, which we will abbreviate as getIA and getIP, that return app bundles, the list of installed apps on a mobile.[3] They can be used by the calling app to find utilities, perform security checks, and other functions. They also have high potential for use in advertising. An illustration of this is the Twitter *app graph* program [46], which was announced in late 2014. Twitter asserted its plans to profile users by collecting their app bundles[4] to "provide a more personal Twitter experience for you." Reacting to Twitter's app graph announcement, the Guardian newspaper postulated [12] that Twitter "reported $320m of advertising revenues in the third quarter of 2014 alone, with 85% of that coming from mobile ads. The more it can refine how they are targeted, the more money it will make." This progression marks an important point about the impact of advertising on privacy. Both the Financial Times [42] and a book about the economics of the online advertising industry called The Daily You [45] emphasize the strong pressures on the advertising industry to deliver better quality information about users in a market place that is both increasingly competitive and increasingly capable. This is a key theme of this paper: what may seem opportunistic now may be accepted business practice and industry standard in a few years, and what is viewed as malicious today may be viewed as opportunistic or adventurous tomorrow. Twitter provides warnings to the user that Twitter will collect app bundles and offers the user a chance to opt out of this. Other parties are less forth-coming about their use of this technique of user profiling.

### A. Use of App Bundles

Getting app bundles is a great illustration of the trajectory of advertising on mobiles. In 2012 the AdRisk tool [18] showed that 3 of 50 representative ad libraries it studied would collect the list of all apps installed on the device. The authors viewed this as opportunistic at best at the time. But what about now? We did a study of the pervasiveness of the use of app bundles by advertising networks in Google Play. The functions getIA and getIP are built into the Android API and require no special permissions. We decompiled the 2700 apps we have collected from Google Play, into smali code [5] for analysis and parsed these files to look for the invocations of getAP and getIP in each app. This allowed us to narrow the set of apps for analysis to only those that actually collect a list of apps on the mobile, which we deem an app bundle. We then conducted a manual analysis of the invocation of these functions by ad libraries.

Of the 2700 apps selected for review, 165 apps were duplicates, narrowing our sample size down to 2535 distinct apps. Of these, 27.5% (679/2535) contained an invocation of either of the two functions. This total includes invocation of these functions for functional (utility and security) as well as advertising purposes. To better understand if an ad library invokes the function, analysis required a thorough examination of the location of the function call to see if it is called by an advertising or marketing library. We found that many apps pass information to advertisers and marketers. We conducted this analysis manually to best capture a thorough list of invocations within ad libraries. Ultimately 12.54% of the examined apps (318/2535) clearly incorporate ad libraries that invoke one of the functions that collects the app bundle of the user. We found 28 different ad libraries invoking either getIA or getIP. These results do not necessarily include those apps that collect app information themselves and pass it to data brokers, advertising or marketing companies, or have their own in-house advertising operation (like Twitter). Our results demonstrate that many types of apps have ad libraries that collect app bundles, including medical apps and those targeted at children. Interestingly, we did not detect collection of app bundles by the three ad networks identified by AdRisk. However, a number of other interesting cases emerged.

`Radio Disney`, for example, uses Burstly, a mobile app ad network whose library [6] calls getIP. Disney's privacy policy makes no direct reference to the collection of app bundles for advertising purposes. Use of this technique in an app targeted at children is troubling because it might collect app bundle information from a child's device without notifying either the parent who assisted the download or an older child that this type of information is collected and used for advertising purposes. Disney does mention the collection of "Anonymous Information" but the broad language defining this does not give any indication that the Radio Disney app collects app bundles.[7]

`Looney Tunes Dash!` is a mobile app provided by Zynga that it explicitly states that they collect "Information

[3] Their formal names are `getInstalledApplications` and `getInstalledPackages`. The first returns the applications, the second returns the packages and, from these, one can learn the application names.

[4] We use the term app bundle rather than app graph because we do not develop a graph from the app lists.

[5] The smali format is a human-readable representation of the application's bytecode.

[6] `burstly/lib/apptracking/AppTrackingManager.smali`

[7] Formally, they define anonymous information as "information that does not directly or indirectly identify, and cannot reasonably be used to identify, an individual guest." App bundles are similar to movie play lists; it is debatable whether they indeed satisfy this definition.

about ... other third-party apps you have on your device."[8] In fact, this is the privacy policy for all Zynga apps.

Several medical apps (12) collect app bundles. Most surprisingly, `Doctor On Demand: MD & Therapy`, an app which facilitates a video visit with board-certified physicians and psychologists collects app bundles through the implementation of google/ads/ conversion tracking. However, their linked privacy policy makes no reference to passing any user information to advertisers. Other apps in the medical category with advertising libraries that collect app bundles include ones that track ovulation and fertility, pregnancy, and remind women to take their birth control pill.

### B. Survey Study

Upon learning of the prevalence of the app bundle collection by advertisers, we sought to better understand what type of information could be learned by advertisers based the list of apps on a user's mobile device. To do this, we devised a study that would allow us to collect our own set of app bundles to train a classifier.

The study consisted of a survey and an Android mobile app launched on the Google Play Store. The protocol for all the parts of the study was approved by the Institutional Research Board (IRB) for our institution. All participants gave their informed consent. We required informed consent during both parts of the study, and participants could leave the study at any time. Participants were informed that the information collected in the survey and the information collected by the mobile app would be associated with one another.

Participants included individuals over the age of 18 willing to participate in the survey and who owned an Android device. Crowdsourcing platforms such as Amazon's Mechanical Turk are proven to be an effective way to collect high quality data [9]. Our survey was distributed over Microworkers.com a comparable crowdsourcing platform to Amazon's Mechanical Turk (MTurk). We chose Microworkers.com over Amazon Mechanical Turk because Amazon Mechanical Turk did not allow tasks that involve requiring a worker to download or install any type of software.

We designed the mobile app, AppSurvey, to collect the installed packages on a participant's phone. The study directed the participant to the Google Play Store to download the mobile app. Upon launching AppSurvey, a pop-up screen provided participants information about the study, information to be collected, and reiterated that the participation in the study was anonymous and voluntary. If the participant declined the consent, no information would be collected. If the participant consented, the app uploaded the app bundles from the participants phone and anonymously and securely transmit it to our server. AppSurvey also generated a unique User ID for each individual which participants were instructed to write down and provide in the survey part of the study. Finally, AppSurvey prompted participants to uninstall the mobile app.

We designed the survey based upon the FT calculator. The survey consisted of 25 questions about basic demographic information, health conditions, and Internet browsing and spending habits. The survey contained two control questions included to identify survey participants not paying sufficient attention while taking the survey. If either of these questions were answered incorrectly, we excluded the survey response. In addition, our workers were not compensated until after the finished tasks were reviewed and approved by the survey conductors. Before taking the survey, participants were required to give informed consent to the information collected in the survey. To link the app bundle information collected by AppSurvey to the responses provided by participants in the survey, participants were required to input the unique User ID generated by AppSurvey. The collection of this data allows us to establish a ground truth for users' app bundles.

We successfully collected survey answers and app bundle information from 243 participants. This resulted in 1985 distinct package names collected.

[8] https://company.zynga.com/privacy/policy

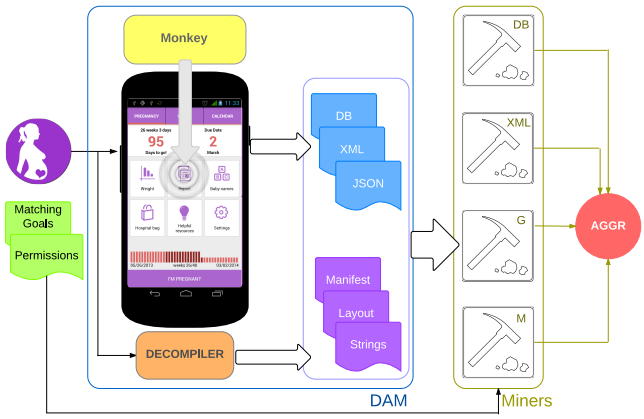



Fig. 2: Design of In-app Pluto

## VI. Pluto: Framework Design and Implementation

Pluto is a modular framework for estimating in-app and out-app targeted data exposure for a given app. In-app Pluto focuses on local files that the app generates, the app layout and string resource files, and the app's manifest file. Out-app Pluto utilizes information about app bundles to predict which apps will be installed together and employs techniques from machine learning to make inferences about users based on the apps they have on their mobile. We describe each of these in a pair of subsections.

### A. In-app Pluto

In-app Pluto progresses in two steps as illustrated in Figure 2. First, the Dynamic Analysis Module (`DAM`) runs the given app on a device emulator and extracts the files the app creates. Then it decompiles the app and extracts its layout files, resource files, manifest file and runtime generated files. At the second step, the files produced by the DAM are fed to a set of file miners. The file miners utilize a set of user attributes and user interests, possibly associated with some domain knowledge, as a matching goal. A miner will reach a matching goal when it decides that a data point is present in a file. When all the app's files are explored, the Aggregator (`AGGR`) removes duplicates from the set of matching goals and the resulting set is presented to the analyst. Pluto's in-app component's goal is to estimate offline, the exposure of targeted data—or data points—to ad libraries at runtime. In-app

Pluto can be configured to estimate data points for a level 1 aggressive library by looking only at the runtime generated files and available permissions. To perform exposure discovery for a level 2 of aggression, it mines targeted data also from the resource and layout files. In essence Pluto is trying to simulate what an ad library is allowed to do to estimate what is the potential data exposure from a given app. To perform in-app exposure discovery, Pluto employs dynamic analysis and natural language processing techniques to discover exposure of in-app data points. Here we report on our prototype implementation focusing on manifest, SQLite, XML, and JSON files.

*1) Dynamic Analysis:* To discover the files that an app is generating at runtime, Pluto runs the app on an emulator for 10 seconds and then uses a monkey tool to simulate user input. [9] This can generate pseudo-random streams of clicks, touches, and system-level events. We chose to use a monkey because some apps might require user stimulation before generating some of their local files. To validate our assumption, we performed two experiments. First, we configured Pluto's DAM module to run all 2535 apps in the FD dataset for 10 seconds each. We repeat the experiment, this time configuring DAM to issue 500 pseudo-random events to each app after its 10 second interval is consumed. As we see on Table III, Pluto explores approximately 5% more apps in the second case. [10] More importantly, DAM_Monkey generates 1196 more files than DAM which results in 100 apps with 'interesting' files more. Android's Monkey was previously found to achieve approximately 25.27% LOC coverage [4]. However, Pluto's components can be easily replaced, and advances in dynamic analysis can be leveraged in the future. For example, PUMA [21] is a very promising dynamic analysis tool introduced recently. If new levels of library aggression are introduced in the future, PUMA could be used instead of Android's monkey to better simulate behaviors that can allow libraries to access user attributes at runtime.

TABLE III: DAM's coverage. * denotes interesting files (SQLite, XML, JSON)

| DA Strategy | % successful experiments | #files | # *files | #of apps w/ *files |
|---|---|---|---|---|
| DAM | 0.718 | 14556 | 9083 | 1911 |
| DAM Monkey | 0.763 | 15752 | 10171 | 2021 |

Once the execution completes, DAM extracts all the 'runtime' generated files. Subsequently, it decompiles the input android app package (apk) and extracts the Android layout files, Android String resources and the app's manifest file.

*2) File Miners empowered by Natural Language Processing:* Once the DAM module generates 'runtime' files, Pluto's enabled file miners commence their exploration. We have implemented four types of file miners in our prototype: `MMiner`; `GMiner`; `DBMiner`; `XMLMiner`. The MMiner is designed to parse manifest files, the DBMiner for SQlite database files, the XMLMiner for runtime generated XML files and the GMiner is a generic miner well suited for resource and layout files. The miners take as input, a set of data points, [11] in the form of noun words and a mapping between permissions and data points that can be derived given that permission.

**Input processing**: Pluto utilizes Wordnet's English semantic dictionary [32] to derive a set of synonyms for each data point. However, a word with multiple meanings will result in synonyms not relevant to Pluto's matching goal. Consider for example the word `gender`. In Wordnet, gender has two different meanings: one referring to grammar rules and the one referring to reproductive roles of organisms. In our case it is clear that we are interested in the latter instead of the former. In our prototype, the analyst must provide Pluto with the right meaning. While it is trivial to make this selection, for other data points it might not be as trivial. For example, `age` has 5 different meanings in Wordnet. Other data points which we have not explored, might have even more complex relationships. In our experience we found *Visuwords.com* to be a helpful tool to visualize such relationships and immensely facilitated our selections. We were inspired by the list of data points in the FT calculator, which is indeed feasible to analyze manually. However, Pluto does not require this from an analyst. If the meaning is not provided, Pluto will take all synonym groups into account with an apparent effect on precision.

**NLP in Pluto**: The NLP community developed different approaches to parse sentences and phrases such as `Parts of Speech` (POS) Tagging and `Phrase and Clause Parsing`. The former can identify parts of a sentence or phrase (i.e., which words correspond to nouns, verbs, adjectives or prepositions), and the latter can identify phrases. However, these cannot be directly applied in our case because we are not dealing with well written and mostly grammatically correct sentences. In contrast, Pluto parses structured data written in a technically correct way (e.g., .sqlite, .xml files). Thus in our case we can take advantage of the well-defined structure of these files and extract only the meaningful words. For the database files, potentially meaningful words will constitute the table name and the columns names. Unfortunately, words we extract might not be real words. A software engineer can choose anything for the table name (or filename), from `userProfile`, `user_profile`, `uProfil`, to `up`. We take advantage of the fact that most software engineers do follow best practices and name their variables using the first two conventions, the `camelCase` (e.g. userProfile) and the `snake_case` structure (e.g. user_profile). The processed extracted words are checked against Wordnet's English semantic dictionary. If the word exists in the dictionary, Pluto derives its synonyms and performs a matching test against the data points and their synonyms. [12] If a match is determined, then a disambiguation layer decides whether to accept or reject the match. Next, we elaborate on the functions of the disambiguation layer.

**Context Disambiguation Layer**: Words that reach a matching goal, could be irrelevant with the actual user attribute. Consider for example the word `exercise`. If a Miner unearths that word, it will be matched with the homonymous synonym of the matching goal `workout`. However, if this word is found in the Strings resource file that doesn't necessarily mean that the user is interested in fitness activities. It could be the case that the app in question is an educational app that has exercises for students. On the other hand, if this word is mined

[9] In our implementation we used the Android SDK-provided `UI/Application Exerciser Monkey` [11].

[10] An unsuccessful experiment includes apps that failed to launch or crashed during the experiment.

[11] We derived most of the data points from the FT calculator [42].

[12] In our prototype we used the JWI [16] interface to Wordnet, to derive sets of synonyms.

from an app in the Health and Fitness Google Play category, then it is more likely this is referring to a fitness activity. Pluto employs a disambiguation layer that aims to determine whether the match is valid. It attaches to every user interest the input app's Google Play category name. We call that a disambiguation term. For user attributes, the disambiguation term is currently assigned by the analyst [13]. In addition, Pluto assigns some domain knowledge to data points. For attributes, it treats the file name or table name as the domain knowledge, and for interests it uses the matching goal itself. Our prototype's context disambiguation layer calculates the similarity between the disambiguation term and the domain knowledge. If the similarity value is found to surpass a specific threshold, then the match is accepted.

The NLP community already proposed numerous metrics for comparing how similar or related two concepts are. Our prototype can be configured to use the following existing similarity metrics to disambiguate attribute matches: PATH [37]; LIN [30]; LCH [28]; LESK [6]. Unlike the first three metrics which are focused on measuring an is-a similarity between two words, LESK is a definition-based metric of relatedness. Intuitively this would work better with user interests where the disambiguation term is the app's category name. The other metrics are used to capture is-a relationships which cannot hold in most of the user-interests cases. For example, there is no strong is-a relationship connecting the user interest vehicle with the category transportation. [14] LESK seems well fit to address this as it depends on the descriptions of the two words. Indeed, LESK scores the (vehicle, transportation) pair with 132 with (vehicle, travel and local) coming second with 103.

However, in this study we have found that LESK might not always work that well when applied in this domain. Studying the scoring of LESK with respect to one of our most popular user interests in our L1 dataset we found it to be problematic. When comparing the matching goal workout with the category Health and Fitness, LESK assigns it one of the lowest scores (33), with the maximum score assigned to the (workout, books and references) pair (113).

Here we present our new improved similarity metric that can address LESK's shortcomings when applied to our problem. We call our similarity metric droidLESK. The intuition behind droidLESK is that the more frequently a word is used in a category, the higher the weight of the (word, category) pair should be. droidLESK is then a normalization of $freq(w, c) \times LESK(w, c)$. In other words, droidLESK is the weighted LESK were the weights are assigned based on term frequencies. To evaluate droidLESK, we create pairs of the matching goal workout with every Google Play category name and assign a score to each pair as derived from droidLESK and other state of the art similarity metrics. To properly weight LESK and derive droidLESK, we perform a term frequency analysis of the workout word in all 'runtime' generated files of the L1 dataset. We repeat the experiment for the word vehicle. droidLESK's scoring was compared with the scores assigned to the pairs by the following similarity metrics: WUP [51]; JCN [24]; LCH [28]; LIN [30]; RES [38]; PATH [37]; LESK [6] and HSO [22].

The results are very promising—even though preliminary—as shown in table IV. [15] We observe that our technique correctly assigns the highest score to the pair (workout, health and fitness) than any other pair (workout,*). The same is true for the pair (vehicle, transportation). droidLesk was evaluated on the two most prevalent user interests in our dataset. Since our approach might suffer from over-fitting, in future work we plan to try this new metric with more words and take into account the number of apps contributing to the term frequency. We further discuss the effects of using droidLESK in Pluto's in-app targeted data discovery in the evaluation Section VII.

### B. Out-app Pluto

Out-app Pluto aims to estimate what is the potential data exposure to an ad library that uses the unprotected public gIA and gIP APIs. That is, given the fact that the ad library can learn the list of installed applications on a device, it aims to explore what data points, if any, can be learned from that list. Intuitively, if an ad library knows that a user installed a pregnancy app and local public transportation app, it would be able to infer the user's gender and coarse location. However, the list of installed applications derived from gIA and gIP is dependent on the device the ad library's host app is installed, which renders estimation of the exposure challenging. To explore what an ad library can learn through this out-app attack channel, we derive a set of co-installation patterns that reveals which apps are usually installed together. This way we can simulate what the runtime call to gIA or gIP will result in given invocation from an ad library incorporated into a particular host app. We then feed the list of co-installed applications into a set of classifiers we trained to discover the potential data exposure through the out-app channel.

The Pluto out-app exposure discovery system runs machine learning techniques on a corpus of app bundles to achieve two goals. First, it provides a Co-Installation Pattern module (CIP) which can be updated dynamically as new records of installed apps are received. The CIP module runs state-of-the-art frequent pattern mining (FPM) algorithms on such records to discover associations between apps. For example, such an analysis can yield an association in the form of a conditional probability, stating that if app A is present on a device then app B can be found on that device with $x\%$ confidence. When an analyst sets Pluto to discover out-app targeted data regarding an app offline, Pluto utilizes the CIP module to get a good estimation of a vector of co-installed apps with the target app. The resulting vector is passed to the classifiers which in turn present the analyst with a set of learned attributes. Second, it provides a suite of supervised machine learning techniques that take a corpus of app bundles paired with a list of user targeted data and creates classifiers that predict whether an app bundle is indicative of a user attribute or interest.

*1) Co-Installation Patterns:* The CIP module uses frequent pattern mining to find application co-installation patterns. This can assist Pluto in predicting what will an ad library learn at runtime if it invokes gIA or gIP. We call a co-installation pattern, the likelihood to find a set of apps installed on a

[13] We used the word Person.

[14] We found that similarity metrics that find these relationships do not assign the best score to the pair(vehicle, transportation) when compared with other (vehicle, *) pairs.

[15] Due to space limitations, we omit uninformative comparisons.

TABLE IV: Comparison between rankings of (interest, category name) pairs from LESK and droidLESK. TF denotes the data point term frequency in local files created by apps in a category.

| DATA POINT | RANK | LESK | TF | TF*LESK |
|---|---|---|---|---|
| VEHICLE | 1 | TRANSPORTATION | FINANCE | TRANSPORTATION |
| VEHICLE | 2 | BOOKS AND REFERENCES | TRANSPORTATION | FINANCE |
| VEHICLE | 3 | TRAVEL AND LOCAL | LIFESTYLE | LIFESTYLE |
| WORKOUT | 1 | BOOKS AND REFERENCES | HEALTH AND FITNESS | HEALTH AND FITNESS |
| WORKOUT | 2 | TRAVEL AND LOCAL | APP WIDGET | NEWS AND MAGAZINE |
| WORKOUT | 3 | MUSIC AND AUDIO | NEWS AND MAGAZINE | APP WIDGET |

device in correlation with another app installed on that device. In FPM, every transaction in a database is identified by an id and an `itemset`. The itemset is the collection of one or more items that appear together in the same transaction. For example, this could be the items bought together by a customer at a grocery store. `Support` indicates the frequency of an itemset in the database. An FPM algorithm will consider an itemset to be frequent if its support is no less than a minimum support threshold. Itemsets that are not frequent are pruned. Such an algorithm will mine `association rules` including frequent itemsets in the form of conditional probabilities that indicate the likelihood that an itemset can occur together with another itemset in a transaction. The algorithm will select rules that satisfy a measure (e.g., a minimum confidence level). An association rule has the form N:N, where N is the number of unique items in the database. An association rule is presented as $X \Rightarrow Y$ where the itemset $X$ is termed the `precedent` and $Y$ the `consequent`. Such analysis is common when stores want to find relationships between products frequently bought together.

Pluto's CIP uses the same techniques to model the installations of apps on mobile devices, as itemsets bought together at a grocery store. Our implementation of Pluto's CIP module uses the FPGrowth [20] algorithm, a state of the art frequent pattern matching algorithm for finding association rules. We have chosen FPGrowth because it is significantly faster than its competitor Apriori [3]. CIP runs on a set of app bundles collected periodically from a database containing user profiles that include the device's app bundles and derives a set of association rules, indicating the likelihood that apps can be found co-installed on a device. Our CIP association rule will have the form 1:N because Pluto is interested in finding relationships between a given app and a set of other apps.

CIP uses `confidence` and `lift` as the measures to decide whether an association rule is strong enough to be presented to the analyst. Confidence is defined as $conf(X \Rightarrow Y) = \frac{supp(X \cup Y)}{supp(X)}$, where $supp(X)$ is the support of the itemset in the database. A confidence of 100% for an association rule means that for 100% of the times that $X$ appears in a transaction, $Y$ appears as well in the same transaction. Thus an association rule $facebook \Rightarrow skype, viber$ with 70% confidence will mean that for 70% of the devices having `Facebook` installed, `Viber` and `Skype` are also installed.

Another measure CIP supports is Lift. Lift is defined as: $lift(X \Rightarrow Y) = \frac{supp(X \cup Y)}{supp(X) \times supp(Y)}$. Lift indicates how independent the two itemsets are in the rule. A Lift of one will indicate that the probability of occurrence of the `precedent` and `consequent` are independent of each other. The higher the Lift between the two itemsets, the stronger the dependency between them and the strongest the rule is.

*2) Learning Targeted Data from App Bundles:* Pluto uses supervised learning models to infer user attributes from the CIP-estimated app bundles. Pluto aims to resolve two challenges in training models based on app bundles: 1) skewed distribution of values of attributes; 2) high dimensionality and highly sparse nature of the app bundles.

**Balancing distributions of training sets:** Based on the empirical data we collected, some attributes have a more skewed distribution in their values. To orient the reader using a concrete example, consider an example where 1 of 100 users has an allergy. In predicting whether a user has an allergy in this dataset, one classifier can achieve an accuracy of 0.99 by trivially classifying each user as having an allergy. In view of this, for the attribute "has an allergy" the value "yes" can be assigned a higher weight, such as 99, while the value "no" has a weight of 1. After assigning weights, the *weighted accuracy* for predicting an attribute now becomes the weighted average of accuracy for each user; the weight for a user is the ratio of the user's attribute value weight to the total attribute value weights of all users. Therefore, in this example, the weighted accuracy becomes 0.5, which is fair, even when trivially guessing that each user has the same attribute value. In order to train an effective model for Pluto, we balance the distribution of training sets following the aforementioned idea. To balance we adjust the weights of existing data entries to ensure that the total weights of each attribute value are equal. In this way, the final model would not trivially classify each user to be associated with any same attribute value. Accordingly, we adopt measures *weighted precision* and *weighted recall* in our evaluation where the total weights of each attribute value are equal; this is to penalize trivial classification to the same attribute value [10].

**Dimension reduction of app-bundle data:** Another challenge we face in this context is the high dimensionality and highly sparse nature of the app bundles. There are over 1.4 million apps [41] on Google Play at this moment, and it is both impractical and undesirable for the users to download and install more than a small fraction of those on their devices. A recent study from Yahoo [39] states that users install on average 97 apps on a device. To make our problem more tractable we used a technique borrowed from the Machine Learning community which allows us to reduce the considered dimensions. Our prototype employs three classifiers, namely K-Nearest Neighbors (KNN), Random Forests, and SVM.

To apply these classifiers to our data, we map each user $u_i$ in the set of users $U$ to an app installation vectors $\mathbf{a}_{u_i} = \{a_1, \ldots, a_k\}$, where $a_j$ = 1 ($j = 1, \ldots, k$) if $u_i$ installs $a_j$ on the mobile device, otherwise $a_j$ = 0. Note that the app installation vector is $k$-dimensional and $k$ can be a large value (1985 in our study). Thus, classifiers may suffer from the "curse of dimension" such that the computation could be dominated by less relevant installed apps when the dimension of space goes higher. To mitigate this problem, we use principal component

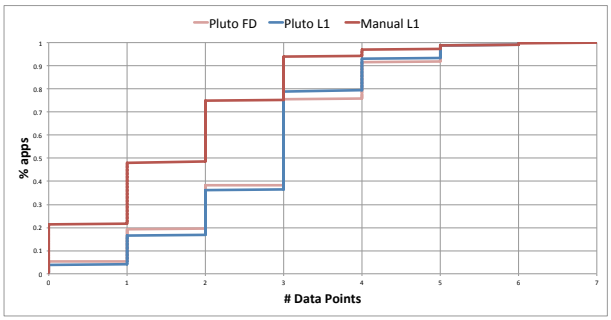



Fig. 3: CDF of apps and number of data points (level-1)

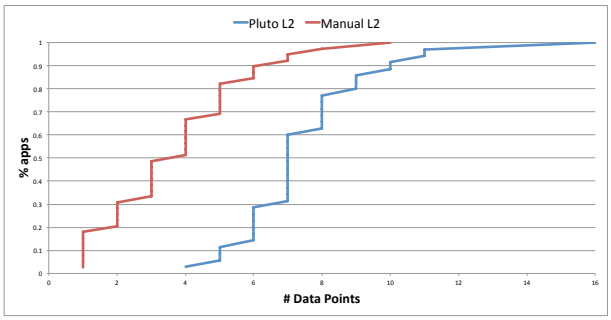



Fig. 4: CDF of apps and number of data points (level-2)

analysis (PCA) by selecting a small number of the principal components to perform dimension reduction before applying a classifier.

## VII. Evaluation

In this section we evaluate Pluto's components in estimating data exposure. We first evaluate Pluto's performance to discover Level-1 and Level-2 in-app data points. Next we apply Pluto's CIP module and classifiers on real world data app bundles and ground truth we have collected, and evaluate their performance.

### A. Evaluation of Pluto's in-app exposure discovery

In this section we present our empirical findings on applying Pluto on real world apps.

**Experimental setup:** We provided Pluto with a set of data points to look for, enhanced with the meaning—sense id of the data point in Wordnet's dictionary—and the class of the data point (i.e., user attribute or user interest). We also provide Pluto with a mapping between permissions and data points and we configured it to use the `LCH` similarity metric at the disambiguation layer for user attributes and our `droidLESK` metric for user interests. We found that setting the LCH threshold to 2.8 and the droidLESK threshold to 0.4 provides the best performance. To tune the thresholds, we parameterized them and ran Pluto multiple times on the L1 dataset. A similar approach can be used to tune the thresholds on any available - ideally larger - dataset [16], and data point set. In all experiments, all Miners were enabled unless otherwise stated. The MMiner mined in manifest files, the DBMiner in runtime-generated database files, the XMLMiner in runtime-generated XML files and the GMiner in String resource files and layout files. We compared Pluto to the level-1 and level-2 ground truth we manually constructed as described in Section IV.

**In-app exposure estimation:** We ran Pluto on the set of 262 apps (Pluto L1) and the full set of 2535 apps (Pluto FD). Figure 3 plots the distribution of apps with respect to data points found within those apps. We saw that the number of data points found in apps remains consistent as we increased the number of apps. We repeated the experiment for the level-1 dataset that consists of 35 apps. Figure 4. depicts Pluto's data point discovery. We compared Pluto's data point prediction with the respective level-1 and level-2 manual analysis IV.

Evidently, Pluto is optimistic in estimating in-app data points. In other words, Pluto's in-app discovery component can flag apps as potentially exposing data points, even though these are not actually there. A large number of Pluto's false positives stem from parsing the String constants file. Parsing these files increases coverage by complementing our dynamic analysis challenge in generating files that host apps created after the user logged in. It also addresses the layer-2 aggressive libraries can read from the user input. However, this results in considering a lot of extra keywords that might match a data point or its synonyms. Their location in the Strings.xml makes it harder for Pluto to disambiguate the context for certain data point classes. In this work, we make the first attempt towards mitigating this pathology by proposing droidLESK.

Pluto is designed to find user attributes, user interests, and data points stemming from the host app's granted permissions. We next present the performance of Pluto's prototype implementation with respect to the above categories.

**Finding user-attributes:** Figure 5 depicts the performance of Pluto in finding the data point `gender` when compared to the level-1 and level-2 datasets and Figure 6 shows the same for the user attribute `age`. Gender had absolute support of 13 in the level-1 dataset and 18 in the level-2 and `age` had 12 and 9 respectively. We observe that Pluto is doing better in discovering data points available to the more aggressive libraries. For example, the word age, was found in a lot of layout files and Strings.xml files while the same was not present in the runtime generated files. Comparing age with the level-1 ground truth, results in a high number of false positives, since the analyst has constructed the ground truth for a level-1 aggressive library. When Pluto is compared with the ground truth for a level-2 aggressive library, its performance is significantly improved.

**Finding interests:** Next, we evaluated Pluto's performance in discovering user interests. Figure 7 illustrates the user interest `workout` when Pluto is compared against the level-1 ground truth and the level-2 ground truth. Workout had absolute support of 5 in the level-1 dataset and 6 in the level-2. Again, Pluto does much better in the latter case for the same reasons stated before.

**Preliminary results for droidLESK:** In our experiments we used droidLESK as the most appropriate similarity metric on Pluto's context disambiguation layer for user interests. We compared that with an implementation of Pluto with no disambiguation layer and an implementation that uses the LESK metric. `droidLESK` achieved an astonishing 103.3% increase in Pluto's precision whereas LESK achieved an improvement of 11.37%. This is a good indication that droidLESK is a promising way of introducing domain knowledge when comparing the similarity between words in the Android app context. We plan to further explore droidLESK's potential in future work.

**Finding data point exposure through permission inheritance:** Pluto's MMiner scrapes through application manifest

[16]Note that it requires little effort to get Android app packages.

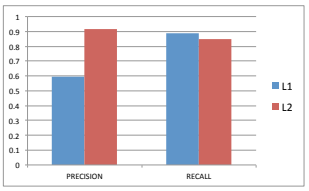



Fig. 5: **gender** prediction performance given the L1 and L2 ground truth.

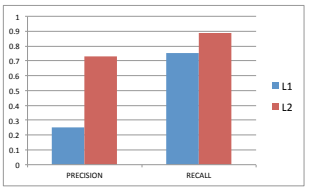



Fig. 6: **age** prediction performance given the L1 and L2 ground truth.

files to look for permissions that would allow a level-1 or level-2 aggressive library to get access to user attributes or interests. We compared Pluto's performance in two different configurations. In configuration 1 (L1 or L2), Pluto is set to look for a data point using all of its Miners whilst in configuration 2 (L1:MMiner and L2:MMiner) Pluto is set to look for a data point only using the MMiner, if the data point can be derived from the host app permissions. We performed the experiment on the larger level-1 dataset, providing as input the mapping between the permissions ACCESS_COARSE_LOCATION and ACCESS_FINE_LOCATION with the data point `address`. Figure 8 depicts Pluto's performance in predicting the presence of `address` given the above two configurations for both the L1 and L2 datasets and ground truths. As expected, Pluto's prediction is much more accurate when only the MMiner is used. It is clear that in the cases where an data point can be derived through a permission, the best way to predict that data point exposure would be to merely look through the target app's manifest file.

The main reason for the false negatives we observed in all previous experiments was because some data points that the analyst has discovered were in runtime files generated after the user has logged in the app, or after a specific input was provided. Pluto's DAM implementation cannot automatically log in the app. We leave this challenge open for future work.

### B. *Evaluation of Pluto's out-app exposure discovery*

Next, we wanted to evaluate Pluto's ability to construct co-installation patterns and predict user attributes and interests based on information that can be collected through the out-app channel. We ran Pluto's CIP module and classifiers on the ABD dataset we collect from real users (see Section V).

**Mining application co-installation patterns:** Our implementation of Pluto's CIP module uses FPGrowth [20], the state of the art frequent pattern matching (FPM) algorithm for finding association rules. We chose FPGrowth because it is significantly faster than its competitor Apriori [3]. We applied Pluto's CIP module on the app bundles we collected through our survey. We set FPGrowth to find co-installation patterns in the form 1:N and prune events with `support` less than 10%. Table V lists the 5 strongest—in terms of `confidence`—association rules that CIP found when run on the survey dataset.

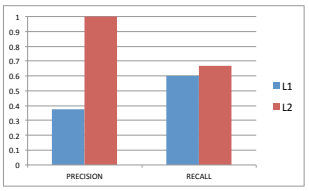



Fig. 7: **workout** prediction performance given the L1 and L2 ground truth.

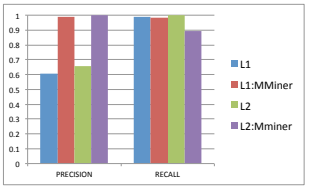



Fig. 8: **address** prediction performance in different configurations, given the L1 and L2 ground truth.

We observe that `Facebook` is likely to be installed together with the `Facebook Messenger` app. This is likely because Facebook asks their users to install the `Facebook Messenger` app when using the Facebook app. Our survey dataset reflects this as well. The strong relationship between the Facebook app and Facebook Messenger app revealed by FPM illustrates its effectiveness for this application. Such rules are critical for Pluto to estimate co-installation patterns between the input application and other applications. Pluto leverages such patterns to provide an estimation of what user attributes can be potentially derived from the app bundles of users that have the input app. Co-installation patterns can also be used to reduce redundancy when combining the in-app data exposure of multiple applications. For example, one might want to estimate what are the in-app data points exposed by app A and app B. However, if these applications are installed on the same device, then the total amount of information the adversarial library will get will be the union of both removing duplicates.

**Performance of Pluto's classifiers:** Pluto's classifiers can be used to estimate user attributes derived from CIP app bundles or real-time app bundles from user profiles. We evaluated the performance of Pluto's classifiers on real app bundles we collected from our survey (see Section V). We used the users' answers to the questionnaire in the survey as the ground truth to evaluate the classification results. To justify our use of dimension reduction technique, we evaluated the classifier on both dataset before dimension reduction and dataset after dimension reduction. The results on representative attributes are shown in Table VI and Table VII respectively.

Based on the results shown in both tables, Random Forest performs best across all prediction tasks. The superiority of Random Forest in our evaluation agrees with the existing knowledge [15]. Specifically, because our dataset has a relatively smaller number of instances, the pattern variance is more likely

TABLE V: The strongest co-installation patterns found by the CIP module when run on the survey app bundles.

| Precedent | Consequence | Conf | Lift |
|---|---|---|---|
| com.facebook.katana | com.facebook.orca | 0.79 | 2.10 |
| com.lenovo.anyshare.gps | com.facebook.orca | 0.75 | 2.01 |
| com.viber.voip | com.facebook.orca | 0.74 | 1.98 |
| com.skype.raider | com.facebook.orca | 0.71 | 1.88 |
| com.skype.raider | com.viber.voip | 0.70 | 2.32 |

TABLE VI: Performance of classifiers before dimension reduction

| Classifier | Age | | Marital Status | | Sex | |
|---|---|---|---|---|---|---|
| | P(%) | R(%) | P(%) | R(%) | P(%) | R(%) |
| Random Forest | 64.1 | 66.3 | 89.8 | 83.6 | 91.5 | 89.6 |
| SVM | 65.5 | 63.6 | 89.0 | 82.1 | 87.4 | 83.1 |
| KNN | 62.7 | 60.0 | 86.3 | 77.7 | 83.4 | 74.8 |

P = Weighted Precision, R = Weighted Recall

TABLE VII: Performance of classifiers after dimension reduction

| Classifier | Age | | Marital Status | | Sex | |
|---|---|---|---|---|---|---|
| | P(%) | R(%) | P(%) | R(%) | P(%) | R(%) |
| Random Forest | 88.6 | 88.6 | 95.0 | 93.8 | 93.8 | 92.9 |
| SVM | 44.8 | 35.4 | 66.9 | 50.5 | 80.9 | 70.1 |
| KNN | 85.7 | 83.6 | 92.5 | 91.2 | 91.6 | 89.9 |

P = Weighted Precision, R = Weighted Recall

to be high. The ensemble technique (voting by many different trees) employed by Random Forest could reduce such variance in its prediction and thus achieve a better performance.

Comparison of Table VI and Table VII show dimension reduction can effectively improve the performance of Random Forest and KNN. However, the performance of SVM becomes poorer after dimension reduction. One possible reason is that SVM can handle high-dimension data such as our original dataset. The model complexity of SVM is determined by the number of support vectors instead of dimensions.

## VIII. Discussion

**Utility of Pluto:** In this work, we propose an approach that can be leveraged to assess potential data exposure through in-app and out-app channels to a third-party library. We note that even though we use ad libraries in free apps as a motivating example, our approach can be adapted to assess data exposure by any app to any third-party library. We chose ad libraries because they are quintessential examples of third-party libraries with strong business incentives for aggressive data harvesting. Motivated by rising privacy concerns related to mobile advertising, users can exert pressure on markets to integrate data exposure assessment into their system and present the results in a usable way to users when downloading an app. In light of this information, users would be able to make more informed decisions when choosing an app. Furthermore, government agencies, such as the Food and Drug Administration (FDA), could benefit from this approach to facilitate their efforts in regulating mobile medical device apps [1] and the Federal Trade Commission (FTC) could leverage Pluto to discover apps that potentially violate user privacy.

We describe a simple way for markets (and in extend other interested parties) to utilize Pluto's results and rank apps based on their data exposure. Intuitively, the harder it is for an adversary to get a data point of a user, the more valuable that data point might be for the adversary. Also, the more sensitive a data point is, the harder it will be to get it. Thus sensitive data points should be more valuable for adversaries. Consequently, a market could use a cost model, such as the one offered by the FT calculator, to assign the proposed values acting as weights to data points. In fact, Google, which acts as a data broker itself, would probably have more accurate values and a larger set of data points. They could then normalize the set of exposed data points and present the data exposure score for each app. For example, let $D$ be the set of data points in the cost model and $X$ the set of data point weights in the cost model, where $|D| = |X| = n$. We include the `null` data point in $D$ with a corresponding zero value in $X$. Also, let $\alpha$ be the app under analysis. Then the new ranked value of $\alpha$ would be $z_\alpha = \frac{x_\alpha - min(X)}{\sum_{i=1}^{n} x_i - min(X)}$ where $x_\alpha$ is the sum of all weights of the data points found to be exposed by app $\alpha$. Here, $min(X)$ corresponds to an app having only the least expensive data point in $D$. $\sum_{i=1}^{n} x_i$ corresponds to an app exposing all data points in $D$. $z_\alpha$ would result in a value from 0 to 1 for each app $\alpha$ under analysis. The higher the value the more the data exposure. This can be presented in the applications download site in application markets along with other existing information for that app. For better presentation, markets could use a number from 0 to 10, stars, or color spectrum with red corresponding to the maximum data exposure.

To provide the reader with a better perspective on the result of this approach, we applied Pluto and performed the proposed ranking technique on the collected apps from the `MEDICAL` and `HEALTH & FITNESS` Google Play categories respectively. In the absence of co-installation patterns for all target apps, we do not take into account the effect of having an app on the same device with another data exposing app [17]. We found that most apps have a low risk score. In particular 97% of `MEDICAL` and also 97% of `HEALTH & FITNESS` apps had scores below 5.0. Those apps either expose a very small amount of highly sensitive targeted data, targeted data of low sensitivity, or both. For example, we found *net.epsilonzero.hearingtest*, a hearing testing app, exposed two attributes, the user's phone number and age, and scored 0.02. This ranking technique ensures that only a few apps stand out in the rankings. These are apps with a fairly large number of exposed data points including highly sensitive ones. For example, the highest scored medical app `com.excelatlife.depression` with a score of 8.14, exposes 16 data points including "depression," "headache," and "pregnancy," which have some of the highest values in the FT calculator. Table VIII depicts the two most risky apps per category. Pluto in conjunction with our ranking approach can help a user/analyst to focus on those high risk cases.

Our ranking results also depict the prevalence of targeted data exposure. As we observe on Table VIII the highest ranked apps were installed in the order of hundreds of thousands of devices. Consequently, highly sensitive data of hundreds of thousands of users are exposed to opportunistic third-party libraries. In future work we plan to study practical approaches to mitigate the data exposure by apps to third-party libraries.

**App Bundles:** The collection of app bundle information by app developers, advertising companies, and marketing companies is troubling. Currently, the ability of apps to use gIP or gIA with no special permissions provides an opportunity for abuse by both app developers and advertisers. Our research demonstrates that this abuse is occurring. We further demonstrate that such information can be reliably leveraged to infer users' attributes. Unfortunately, companies

[17] Note that to perform the out-app Pluto analysis one needs co-installation patterns for all ranked apps. Markets can easily derive those using our FPM approach. In that case, one should take into account the UNION of in-app and out-app exposed attributes.

TABLE VIII: Most risky apps based on their in-app data exposure. M = MEDICAL, HF = HEALTH & FITNESS

| CATEGORY | PACKAGE | DESCRIPTION | AVG #INSTALL | SCORE [0 - 10] |
|---|---|---|---|---|
| M | com.excelatlife.depression | Depression management | $100 \times 10^3 - 500 \times 10^3$ | 8.14 |
| M | com.medicaljoyworks.prognosis | Clinical case simulator for physicians | $500 \times 10^3 - 1 \times 10^6$ | 6.31 |
| HF | com.workoutroutines.greatbodylite | Workout management | $100 \times 10^3 - 500 \times 10^3$ | 7.33 |
| HF | com.cigna.mobile.mycigna | Personal health information management | $100 \times 10^3 - 500 \times 10^3$ | 5.62 |

fail to notify consumers that they are allowing the collection of app bundles. With this, they have also failed to notify users as to what entity collects the information, how it is used, or steps to mitigate or prevent the collection of this data. The failure of the Android API to require permissions for the gIP or gIA removes from the users the possibility to have choice and consent to this type of information gathering. To prevent abuse of gIP or gIA, app providers should notify users, both in the privacy policy and in the application, that app bundles are collected. Additionally, applications should provide the user the opportunity to deny the collection of this information for advertising or marketing purposes. Potentially, the Android API could require special permissions for gIP or gIA. However, the all-or-nothing permissions scheme might not add any additional value besides notice to the user and the warning may not be necessary for an app that is using these two functions for utility and functional purposes.

**Limitations of our approach:** Our estimation of data exposure to libraries is constrained by the specific attack channels we consider. Our prototype employs specific examples for each channel and performs data exposure assessment based on those. Nevertheless, the cases we consider are not the only ones. For example, someone could include the `CAMERA` permission or the `RECORD_AUDIO` in the protected APIs. The camera could be used opportunistically to get pictures of the user in order to infer her gender or location. The microphone could be used to capture what the user is saying and, by converting speech to text and employing POS tagging, infer additional targeted data. More channels can also be discovered such as new side channels or covert channels. These can be used to extend Pluto for a more complete assessment. Our current prototype and results can serve as a baseline for comparison.

## IX. Related Work

Several efforts try to characterize the current mobile ad targeting process. MAdScope [35] and Ullah et al. [47] both found that ad libraries have not yet exploited the full potential of targeting. Our work is driven by such observations and tries to assess the data exposure risk associated with embedding a library in an app.

Many studies describe alternative mobile advertising architectures. AdDroid [36] enforces privilege separation by hard-coding advertising functions as a system service into Android platform. AdSplit [40] achieves privilege separation via making ad libraries and their host apps run in separate processes. Leontiadis et al. [29] proposes a client-side library compiled with the host app to monitor the real-time communication between the host app and the ad libraries to control the exposed information. MobiAd [19] suggests local profiling instead of keeping the user profiles at the data brokers to protect users' privacy. Most of these alternative architectures envision a separation of ad libraries from their host apps. This would eliminate the in-app attack channels that we demonstrate and constrain the data exposure to the ad libraries. However, none of these solutions are deployed in practice as they all disrupt the business model of multiple players in this ecosystem. We take a different approach by modeling the capabilities of ad libraries in order to proactively assess apps' data exposure risk.

There are a number of studies that aim to—or can be used to—detect and/or prevent current privacy-infringing behaviors in mobile ads. Those works mainly fall into three general categories: (1) static scanning [18], [17], [31], [5], [8], (2) dynamic monitoring [43], [52], [48], [14], and (3) hybrid techniques using both [33]. A combination of these techniques could detect and prevent some of the attack strategies of ad libraries we discussed in this work, if they are adopted in practice. However, such countermeasures can still fail to protect against all allowed behaviors. For example, TaintDroid [14] and FlowDroid [5] cannot evaluate the sensitivity of the data carried. Moreover, static code analysis will miss dynamically loaded code, and code analysis in general cannot estimate the potential reach of libraries. Further, by merely encrypting local files we cannot prevent libraries within the same process from using the key the host app uses to decrypt the files. In addition, there is no mechanism to address data exposure through app bundle information as we reveal in this work because (1) this is not considered as a sensitive API from AOSP and (2) even if marked as sensitive it is unclear how access to it by apps and/or libraries should be mediated, as there are legitimate uses of it. Our focus is not on detecting and tackling current behaviors but assessing the data exposure given the allowed behaviors. This is critical when trying to assess the privacy risk of an asset.

SUPOR [23] and UIPicker [34] seek instances where apps exfiltrate sensitive data. Like Pluto, they use NLP and machine learning techniques to find data of interest in user interfaces. Unlike Pluto, their focus is on data like account credentials and financial records, whereas Pluto is aimed at general targeted data with validation based on data of interest to advertisers. As with most of the other work in this area, SUPOR and UIPicker seek existing exfiltration instances rather than allowed instances, although some of their techniques can facilitate finding allowed instances.

## X. Conclusion

In this work, we studied the feasibility and security implications of fully exploring advertising libraries' capabilities in Android apps. We manually investigated the prevalence of targeted data exposure and revealed a trend in ad networks to become more aggressive towards reachable user information. We designed, implemented, and evaluated Pluto, a modular framework for privacy risk assessment of apps integrating ad libraries. We show that Pluto can be used for automatic detection of targeted data exposure in Android apps. We hope that our work will inspire related attempts to systematically assess the data exposure to ad libraries and that Pluto will serve as a baseline in evaluating future frameworks.

## ACKNOWLEDGMENTS

This work was supported in part by HHS 90TR0003-01, NSF CNS 13-30491, NSF CNS 12-23967 and NSF CNS 15-13939. The views expressed are those of the authors only. The authors are grateful to Hari Sundaram for his comments on advertising, Andrew Rice for sharing the Device Analyzer [49] dataset, and NDSS shepherd, Venkat Venkatakrishnan for his valuable assistance in improving the final version of this paper.

## REFERENCES

[1] Fda.gov. http://goo.gl/guSsMM. Accessed: 2015-01-05.

[2] Pluto code base and experimental results. https://goo.gl/dxXI4O, Observed in May 2015.

[3] R. Agrawal, R. Srikant, et al. Fast algorithms for mining association rules. In *VLDB*, 1994.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *ACM ASE*, 2012.

[5] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI*, 2014.

[6] S. Banerjee and T. Pedersen. An adapted lesk algorithm for word sense disambiguation using wordnet. In *CICLing*. 2002.

[7] P. Barford, I. Canadi, D. Krushevskaja, Q. Ma, and S. Muthukrishnan. Adscape: Harvesting and analyzing online display ads. In *WWW*, 2014.

[8] R. Bhoraskar, S. Han, J. Jeon, T. Azim, S. Chen, J. Jung, S. Nath, R. Wang, and D. Wetherall. Brahmastra: driving apps to test the security of third-party components. In *USENIX Security*, 2014.

[9] M. D. Buhrmester, T. Kwang, and S. D. Gosling. Amazon's Mechanical Turk: A new source of inexpensive, yet high-quality data? *Perspectives on Psychological Science*, 2011.

[10] J. Carroll and T. Briscoe. High precision extraction of grammatical relations. In *COLING*, 2002.

[11] developer.android.com. Ui/application exerciser monkey. http://goo.gl/cH9wPR, Observed in May 2015.

[12] S. Dredge. Twitter scanning users' other apps to help deliver 'tailored content'. *The Guardian*, November.

[13] C. Duhigg. How companies learn your secrets. *The New York Times*, 16, 2012.

[14] 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 *OSDI*, 2010.

[15] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim. Do we need hundreds of classifiers to solve real world classification problems? *J. Mach. Learn. Res.*, Jan. 2014.

[16] M. A. Finlayson. Java libraries for accessing the princeton wordnet: Comparison and evaluation. In *GWC*, 2014.

[17] C. Gibler, J. Crussell, J. Erickson, and H. Chen. Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *TRUST*, 2012.

[18] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *WiSec*, 2012.

[19] H. Haddadi, P. Hui, and I. Brown. Mobiad: private and scalable mobile advertising. In *MobiArch*, 2010.

[20] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[21] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan. Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Mobisys*, 2014.

[22] G. Hirst and D. St-Onge. Lexical chains as representations of context for the detection and correction of malapropisms. *WordNet: An electronic lexical database*, 1998.

[23] J. Huang, Z. Li, X. Xiao, Z. Wu, K. Lu, X. Zhang, and G. Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *USENIX Security*, 2015.

[24] J. J. Jiang and D. W. Conrath. Semantic similarity based on corpus statistics and lexical taxonomy. *arXiv.org*, 1997.

[25] Y. Jing, G. Ahn, Z. Zhao, and H. Hu. Towards automated risk assessment and mitigation of mobile application. *Dependable and Secure Computing, IEEE Transactions on*, 2014.

[26] Y. Jing, G.-J. Ahn, Z. Zhao, and H. Hu. Riskmon: Continuous and automated risk assessment of mobile applications. In *CODASPY*, 2014.

[27] S. Kaplan and B. J. Garrick. On the quantitative definition of risk. *Risk Analysis*, 1981.

[28] C. Leacock and M. Chodorow. Combining local context and wordnet similarity for word sense identification. *WordNet: An electronic lexical database*, 1998.

[29] I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo. Don't kill my ads!: balancing privacy in an ad-supported mobile application market. In *HotMobile*, 2012.

[30] D. Lin. An information-theoretic definition of similarity. In *ICML*, 1998.

[31] C. Mann and A. Starostin. A framework for static detection of privacy leaks in Android applications. In *SAC*, 2012.

[32] G. A. Miller. Wordnet: a lexical database for english. *CACM*, 38(11), 1995.

[33] V. Moonsamy, M. Alazab, and L. Batten. Towards an understanding of the impact of advertising on data leaks. *IJSN*, 7(3), 2012.

[34] Y. Nan, M. Yang, Z. Yang, S. Zhou, G. Gu, and X. Wang. Uipicker: User-input privacy identification in mobile applications. In *USENIX Security*, 2015.

[35] S. Nath. MAdScope: Characterizing mobile in-app targeted ads. In *Mobisys*, 2015.

[36] P. Pearce, A. P. Felt, G. Nunez, and D. Wagner. Addroid: Privilege separation for applications and advertisers in Android. In *ASIACCS*, 2012.

[37] R. Rada, H. Mili, E. Bicknell, and M. Blettner. Development and application of a metric on semantic nets. *IEEE SMC*, 1989.

[38] P. Resnik. Using information content to evaluate semantic similarity in a taxonomy. *arXiv.org*, 1995.

[39] P. Sawers. businessinsider.com. http://goo.gl/8g34xB, Observed in May 2015.

[40] S. Shekhar, M. Dietz, and D. S. Wallach. AdSplit: Separating smartphone advertising from applications. In *USENIX Security*, 2012.

[41] D. Smith. businessinsider.com. http://goo.gl/LNn0pi, Observed in May 2015.

[42] E. Steel, C. Locke, E. Cadman, and B. Freese. How much is your personal data worth? *Financial Times*, June 2013.

[43] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in Android ad libraries. In *MoST*, 2012.

[44] L. Sweeney. K-anonymity: A model for protecting privacy. *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, 2002.

[45] J. Turow. *The daily you: How the new advertising industry is defining your identity and your worth*. Yale University Press, 2012.

[46] Twitter. What is app graph in Twitter? https://goo.gl/scmc69, Observed in May 2015.

[47] I. Ullah, R. Boreli, M. A. Kaafar, and S. S. Kanhere. Characterising user targeting for in-app mobile ads. In *INFOCOM WKSHPS*, 2014.

[48] L. Vigneri, J. Chandrashekar, I. Pefkianakis, and O. Heen. Taming the Android appstore: Lightweight characterization of Android applications. *arXiv.org*, 2015.

[49] D. T. Wagner, A. C. Rice, and A. R. Beresford. Device analyzer: Understanding smartphone usage. In *MobiQuitous*, 2013.

[50] C. E. Wills and C. Tatar. Understanding what they do with what they know. In *WPES*, 2012.

[51] Z. Wu and M. Palmer. Verbs semantics and lexical selection. In *ACL*, 1994.

[52] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang. AppIntent: Analyzing sensitive data transmission in android for privacy leakage detection. In *CCS*, 2013.

[53] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming information-stealing smartphone applications (on android). In *TRUST*, 2011.