2014 IEEE NINTH INTERNATIONAL CONFERENCE ON INTELLIGENT SENSORS, SENSOR NETWORKS AND INFORMATION PROCESSING (ISSNIP)
SYMPOSIUM ON SECURITY, PRIVACY AND TRUST FOR CYBER-PHYSICAL SYSTEMS
SINGAPORE, 21–24 APRIL 2014

AdDetect: Automated Detection of Android Ad Libraries using Semantic Analysis

Annamalai Narayanan, Lihui Chen, and Chee Keong Chan
School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore.
annamala002@e.ntu.edu.sg, {elhchen, eckchan}@ntu.edu.sg

***Abstract*—Applications that run on mobile operating systems such as Android use in-app advertisement libraries for monetization. Recent research reveals that many ad libraries, including popular ones pose threats to user privacy. Some aggressive ad libraries involve in active privacy leaks with the intention of providing targeted ads. Few intrusive ad libraries are classified as adware by commercial mobile anti-virus apps.**

Despite such issues, semantic detection of ad libraries from Android apps remains an unsolved problem. To this end, we have proposed and developed the AdDetect framework to perform automatic semantic detection of in-app ad libraries using semantic analysis and machine learning. A module decoupling technique based on hierarchical clustering is used to identify and recover the primary and non-primary modules of apps. Each of these modules is then represented as vectors using semantic features. A SVM classifier trained with these feature vectors is used to detect ad libraries. We have conducted an experimental study on 300 apps spread across 15 categories obtained from the official market to verify the effectiveness of AdDetect. The simulation results are promising. AdDetect achieves 95.34% accurate detection of ad libraries with very less false positives. Further analysis reveals that the proposed detection mechanism is robust against common obfuscation techniques. Detailed analysis on the detection results and semantic characteristics of different families of ad libraries is also presented.

I. Introduction

Google's Android is a popular Linux based mobile operating system designed to download and run applications (apps, for short) which are predominantly written in Java and distributed as compressed application package (apk) files through official and third party markets. Android offers a sandboxed permission based security model, where, every app must statically request for permissions to access sensitive data and resources at the time of installation. On execution, every app runs as a separate Linux process with a unique UserID in mutual isolation from other concurrently running apps. Reports and statistics such as download count reveal that the free apps are more popular than the paid ones, and they account to more than two thirds of the total number of apps in the official market called Google Play [1].

The Android ecosystem nurtures vibrant advertising through a large number of advertisement agencies providing ad libraries and help the developers monetize their apps. Ad servers maintained by ad agencies host the ad content provided by multiple advertisers and deliver them upon request to the apps. The developers upon registration with the ad networks are provided with a precompiled Software Development Kit (SDK), which is bundled with the app to facilitate ad delivery. On execution, the in-app ad library sends requests to the ad server, receives and parses the responses and subsequently, takes care of displaying the ad content and handling user interaction. Thus, from a developer's viewpoint, in-app ad libraries abstract away the details of the ad delivery process. Developers use multiple ad libraries in their apps to increase the ad revenue. Similarly, Android apps, depending on their category, use many other types of libraries such as social SDKs, game engines, in-app billing libraries etc.

TABLE I
Previous research work using whitelist based approach to detect ad libraries in Android apps

Previous Work	Goal of the research work	No. of ad libraries used in whitelist
AdRisk[2]	Detection of privacy leaks through ad libraries	100
Stevens et al. [3]	Analysis on privacy threats posed by ad libraries and their permission usage	13
Book et al. [4]	Longitudinal analysis on behavior of ad libraries	68
AdDroid [5]	Privilege separation between ad library and host app	9
Pearce et al.[6]	Profiling apps based on ad library usage	30
Book et al. [7]	Privacy leak focused study on interface between ad libraries and host apps	103
AdRob [8]	Study on loss of ad revenue and related impacts due to app plagiarism	16

Prominent recent research on Android ad libraries is presented in Table I. While the goals of these studies are to demonstrate the analysis that happens on ad libraries and not the detection of ad libraries itself, all of them follow a *whitelist* based approach to detect ad libraries. That is, after reverse engineering the apps, a string search is performed with the name of ad SDK package to ensure the usage of a particular ad library. For example, the presence of *admob* ad library is ensured if the package *admob/android/ads* is found in a reverse engineered app. *AdRisk* [2] uses the names of 100 widely used ad SDK packages to detect their presence in apps and for further analysis on privacy threats. As indicated in Table I, studies [2-8] use whitelist based approach to ensure the presence of certain ad libraries. Further more, some popular commercial ad blocker apps also use approaches similar to whitelist based detection. Whitelist based detection of ad libraries has the following drawbacks,

- Given the number of ad supported apps and the vibrant

978-1-4799-2843-9/14/$31.00 © 2014 IEEE

and growing Android advertising ecosystem with a large number of ad network providers, it is not possible to build and maintain a comprehensive whitelist of all the possible ad package names, making this approach unsuitable as a generic solution to detection of ad libraries.
- This approach is not guaranteed to find all ad libraries in an app as detection is confined only to the libraries in the whitelist. [2], [4] and [6] highlight this limitation and acknowledge that their analysis is incomplete because of whitelist based ad library detection.
- Obfuscating or changing the names of the ad library packages would fail this detection process and any further analysis. This is more evident from the fact that popular aggressive push ad libraries such as *airpush* and *airad* have chosen to obfuscate their package names to defeat this kind of detection. Similarly, changing the ad package names for other reasons such as SDK version migration, would fail detection and subsequently fail all types of analysis done in [2-8].

Hence an unconfined robust alternative to whitelist based approach to detection of ad libraries is required. To this end, we propose AdDetect, an automated ad library detection framework based on semantic analysis using machine learning.

Every Android app has a primary module that defines the functionality of the app and other non primary modules (usually libraries). To detect ad libraries, the app has to be first decoupled into individual modules. AdDetect uses hierarchical clustering to perform module decoupling. Semantic features are extracted from each of the modules and are represented as vectors in feature space. Subsequently, a Support Vector Machine (SVM) classifier is built to detect ad libraries from these individual packages.

The main contributions in this paper are as follows,

- A framework to perform automated semantic detection and recovery of ad libraries from Android apps using clustering and classification.
- An improved module decoupling technique with new features to extract the primary and non primary modules of apps.
- Identification of semantic features to detect ad libraries and the corresponding representation of individual modules of apps in the feature space.

II. System Overview

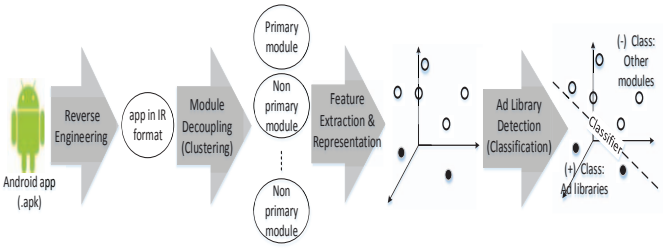

Fig. 1. AdDetect: Overview

The system overview of AdDetect framework is presented in Fig. 1. AdDetect's semantic detection of ad libraries is a 4 step process as explained below,

1) *Reverse Engineering*: Android apps written in Java (might include native code in C/C++) are compiled as *.dex* files designed to run on Davlik Virtual Machine (DVM). The *.dex* files retain rich semantic information that makes sophisticated reverse engineering of apps possible. Analysis of reverse engineered Android apps at the intermediate representation (IR) code level and at Java code level could be found extensively in literature [2-5]. We use *apktool*[9] to obtain the *smali*[10] code (a popular IR code) and the architecture information i.e. packages present in apps.
2) *Module Decoupling*: The core functionality of the app is contained in the primary module and other functionalities are rendered by relatively independent modules such as ad libraries, social SDKs, analytic libraries etc. Based on the architecture information obtained through reverse engineering, a Package Dependency Graph (PDG) is constructed. Hierarchical Agglomerative Clustering (HAC) [13] is applied on the PDG to identify and recover modules of the app. The detailed procedure along with the module decoupling algorithm is presented in section III.
3) *Feature Extraction and Representation*: Feature extraction and application of machine learning techniques have been successfully demonstrated in related topics such as Android malware analysis, vulnerability detection, app vetting and plagiarism detection [12]. To the best of our knowledge, AdDetect is the first approach to perform semantic analysis with the view of detecting ad libraries. To this end, we have extracted semantic features (such as usage of Android components, permissions and APIs) from each of the decoupled modules and represent them as vectors in feature space. This is explained in detail in section IV.
4) *Ad Library Detection*: A classifier is built with these feature vectors and used further to perform automated semantic detection of ad libraries from apps.

III. Module Decoupling

Once the *smali* code of the app is obtained using *apktool*, modules of the app have to be identified and decoupled. We leverage on the fact that modules (Java packages) exhibit high intra-modular and low inter-modular program dependency and apply HAC to recover individual modules. We have improved the module decoupling technique proposed in *PiggyApp* [12], which is a study on plagiarism amongst Android apps.

As explained below, module decoupling involves three main steps: recovering the package hierarchy of the app and identifying the representative packages, estimating the dependencies among the representative packages and construction of PDG, applying hierarchical clustering on PDG to recover the individual modules.

Identifying representative packages: It could be noted that, after obtaining the *smali* code (or any other IR code),

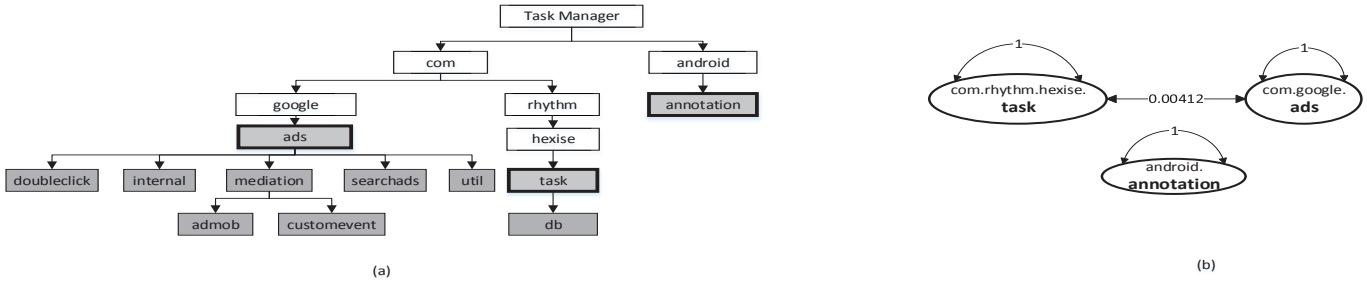

Fig. 2. **(a)**: Packages obtained by reverse engineering com.rhythm.hexise.task.apk using *apktool* **(b)**: Package Dependency Graph

the packages and the hierarchy of the packages present in the original app could be recovered, as this information is retained in the *.dex* files [12]. We use *apktool* to recover the packages of apps and their hierarchy and illustrate this with an example. Fig. 2 (a) depicts the packages and package hierarchy recovered from a popular app called *Task Manager* (MD5: 3377f8527479ab4e72bf9fa5eec62abe). The gray blocks are the Java packages. It is evident that *com.google.ads.mediation.admob* is a sub package of *com.google.ads.mediation*, which in turn is a sub package of *com.google.ads* which happens to be root of this package subtree. Similar assumption holds for other two subtrees in the figure. While analyzing package dependencies to determine the modules of app, considering the root of every package subtree (in the example, *com.google.ads, com.rhythm.hexise.task* and *android.annotation* which are highlighted by bolder blocks) as representative packages is sufficient and would lead to computationally less expensive and quicker convergence compared to considering all the Java packages.

Estimating package dependencies & constructing PDG: Packages at the root of every package subtree in the reverse engineered app form the nodes of the PDG. Weights along the edges in PDG indicate the dependency between packages. Inter class and inter package dependency relationships as listed in Table II are used as features to determine the dependency between packages. Since these features have varying importance in determining the dependency, we distinguish them with empirically determined weights. Relationships indicating stronger dependencies are assigned larger weights. For constructing PDG, intra package dependencies are not considered and external dependency values of a package is normalized by its total external dependency. Hence the dependency between any two packages is in the range [0,1]. For the app discussed as example, we arrive at the PDG as shown in Fig. 2 (b).

Hierarchical Agglomerative Clustering: Performing hierarchical clustering on the PDG with an empirically determined cut off threshold of 0.15, would result in identification of all the modules contained in the app. In the example, three modules namely, *com.google.ads, com.rhythm.hexise.task* and *android.annotation*, are recovered using this procedure.

Algorithm 1 outlines the process of module decoupling. *smali* code and the package hierarchy of the app are obtained using *apktool* (line 1). The packages at the root of every package subtree are identified (lines 2 & 3). The PDG with these packages as nodes is constructed by analyzing the package dependencies (line 4). Finally, the PDG is represented as relational matrix and clustered using HAC, leading to recovery of all the modules contained in the app (line 5).

[1]hierarchical parent-child or sibling relationships among packages

TABLE II
FEATURES USED TO DETERMINE PACKAGE DEPENDENCY & CORRESPONDING WEIGHTS

Feature	Weight
Member field references	1
Method invocation	2
Inheritance	10
Cyclic dependency	10
Package homogeny[1]	10

Algorithm 1 Module Decoupling Algorithm

Input: apk - *.apk* file of Android app
Output: M - Set of primary and non primary modules of the app, $|M|\epsilon\mathbb{N}$

1: ir_code ← *reverse_engineer*(*apk*)
2: T← Construct the package hierarchy tree with ir_code
3: R ← Packages at the root of every package subtree in T
4: PDG ← *construct_package_dependency_graph*(R)
5: M ← *hierarchical_agglomerative_cluster* (PDG)
6: return M

We have made the following improvements in module decoupling with respect to *PiggyApp* [12],

- *PiggyApp* considers every Java package as a node in PDG and analyzes dependencies during module decoupling. Our approach reduces this to considering only the packages that represent the root of package subtrees. For the app considered in example, *PiggyApp* produces a PDG with 11 nodes corresponding to all Java packages in the app, whereas, our approach uses only 3 nodes, leading to faster and computationally less expensive recovery of modules.
- Packages with cyclic dependencies are more likely to be parts of the same module, as cyclic dependency is considered as an anti-pattern for software package design. Hence, we have introduced cyclic dependency as a feature to estimate package dependencies.

TABLE III
EXTRACTED FEATURES - USED IN REPRESENTATION OF MODULES AS FEATURE VECTORS

Feature Set	Features
S_1: App info, $\|S_1\|$ = 18	Usage of activity Usage of service Usage of content providers Usage of broadcast receivers Usage of intents Usage of native code Usage of reflection Usage of dynamic code Usage of static http url Usage of IMEI (UDID) Usage of MEID (UDID) Usage of ESN (UDID) Usage of ANDROID_ID (UDID) Usage of device serial number Usage of device model information Usage of profile information Usage of app's Android package name Is primary package
S_2: Permissions, $\|S_2\|$ = 130	Android Permissions used by the module
S_3: Android APIs, $\|S_3\|$ = 428	Selected critical Android APIs used by the module

These improvements impact the results of module decoupling process positively, as discussed later in section V. After recovering all the modules of the app through aforementioned algorithm, primary module could be identified using the meta information provided in *AndroidManifest.xml*.

IV. FEATURE EXTRACTION AND REPRESENTATION

Once the modules in the app are recovered, we proceed towards representing every module as a vector in feature space and build a SVM classifier to detect ad libraries. Androgurad [11], a tool widely used in analyzing reverse engineered Android apps is deployed in our semantic analysis. The semantic features extracted are listed in Table III.

A. Feature Set S_1: Usage of app component details, device identifiers and users' profile information

The feature set S_1 containing 18 features, pertains to the usage of four types of Android components (i.e. activities, services, content providers and broadcast receivers), intents, device specific information and user profile information. As discussed in [1-3], most of the ad libraries use Unique Device IDentifiers (UDIDs) such as IMEI and device serial number to facilitate identification of unique end user devices at the ad provider's end. This helps the ad providers build profiles of end users and to serve better targeted ads. Though, Google discourages using such UDIDs and promotes the usage of ANDROID_ID to safely identify unique devices without raising privacy concerns, it is reported that many ad libraries still use former device specific artifacts [3]. Hence usage of these UDIDs and ANDROID_ID are considered as features in our method. Also, many ad libraries are found to share the apps' package name along with the users' personal profile information such as location, age and gender (accessible through various means in mobile device) so that targeted ads could be delivered [2-4]. Usage of static http urls and dynamic code loading are considered as important characteristics of most ad libraries, as they use static urls to reach ad servers and download ad content dynamically and run [2], [3]. Also, information such as usage of native code and reflection reveal the characteristics of a module.

B. Feature Set S_2: Usage of Android Permissions

Android provides 130 permissions and governs access to critical resources through them. Permission usage pattern could reveal the characteristic of a module. For instance, all ad libraries require INTERNET permission to facilitate ad request and delivery, whereas a game engine library does not need this permission at all. Studying the apps' permission usage pattern is a popular technique that is successfully adopted in tasks such as detecting malware, vulnerabilities and code clones among Android apps [12]. Hence, the permissions used by modules are considered as features in our approach.

PScout [14] provides mechanisms to identify the permissions used in specific portions of source code by mapping Android APIs to permissions. We leverage on this mapping to identify the permission used by every module.

C. Feature Set S_3: Usage of selected Android APIs

Use of Android APIs that provide access to critical system resources (such as GPS, camera, audio recording, WIFI etc.) are restricted only to apps that have sufficient permissions. Usage of such critical Android APIs certainly provide information on the resources being accessed by the module and thus reveal the characteristics of the module [12]. 428 such critical Android APIs are identified and usage pattern of these APIs are considered as features in our approach.

D. Feature Space Embedding

With features extracted from individual modules, we proceed towards representing each of these modules as vectors in the feature space. A $|S|$-dimensional vector space is defined with the set S, which is,

$$S = S_1 \cup S_2 \cup S_3$$

Every module *m* in the app is mapped to this space by constructing a vector $\phi(m)$, such that for each feature *s* extracted from *m*, the respective dimension is set to 1 and all other dimensions are 0. Map ϕ can be defined for an app with a set of modules *M* as follow,

$$\phi : M \to \{0,1\}^{|S|}, \phi(m) \mapsto (I(m,s))_{s \in S}$$

where the indicator function $I(m,s)$ is defined as,

$$I(m,s) = \begin{cases} 1, & \text{if module } m \text{ contains feature } s \\ 0, & \text{otherwise} \end{cases}$$

As it could be inferred from Table III, $|S|$ = 576.

After obtaining the feature vectors representing individual modules of all the apps in our dataset, we manually labeled them with corresponding classes: ad libraries (positive class)

and other modules (negative class). A SVM classifier is subjected to training and 10-fold cross validation and used in AdDetect framework for further detection of ad libraries from the test set apps.

V. Prototyping and Evaluation

We implemented a prototype of AdDetect framework in Linux. The prototype included *apktool* and *androguard* in module decoupling and feature extraction phases respectively. Weka [15], a well known machine learning workbench is used for classification.

A. Dataset

The dataset used in our experiments is formed by 300 free apps spread across 15 categories, obtained from Google Play [1] in September 2013[2]. We have used apps having no ad libraries to apps having as many as 12 ad libraries in our dataset. After module decoupling, 4777 modules are found to be present in the dataset out of which 563 were ad libraries. Libraries from 79 different ad networks (including popular ones like *admob, inmobi, millinealmedia,* etc. and least used ad libraries such as *ubermind, innoance* and *admozi*) are found among the 563 ad libraries.

B. Module Decoupling

We manually analyzed the results of module decoupling and found that the modules of 284 apps out of 300 apps are properly recovered. This accuracy is achieved using lesser number of Java packages as data items to be clustered compared to *PiggyApp* [12], as depicted in Fig. 3. Due to space constraints, a comparison on number of packages used as data elements to be clustered in the module decoupling phase of the first fifty apps from our dataset is presented in Fig. 3.

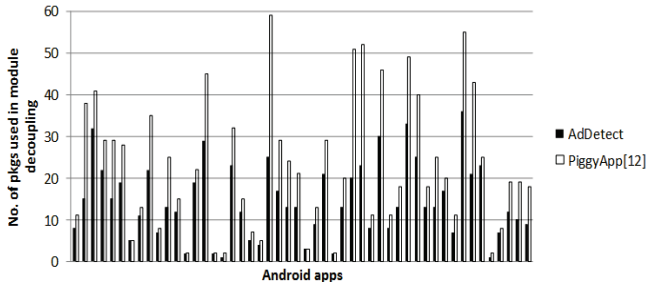

Fig. 3. Comparison - Improved module decoupling proposed in AdDetect Vs. *PiggyApp* [12]

C. Ad Library Detection

Modules of test set apps are supplied to AdDetect's SVM classifier and are tested for ad library detection. Standard evaluation metrics such as precession, recall and F-measure have been used in our experiments. We achieve 95.34 % accurate detection of ad libraries using AdDetect as depicted in as depicted in Table IV (row 1).

[2]The details of the app package name, category, hash codes could etc. be found at: https://sites.google.com/site/annamalainarayanansg/resources

TABLE IV
Results of ad library detection using AdDetect

Dataset	Precision	Recall	F measure
4777 modules	0.953	0.954	0.9534
4777 modules (with SMOTE)	0.958	0.958	0.958
1000 modules (500 ad libraries, 500 non ad library modules)	0.942	0.941	0.9414

In most of the apps, number of non ad library modules outnumber the ad libraries. This is a natural imbalance in the composition of apps and this is observed in our experimental dataset as well. To evaluate the performance of detection process without this imbalance, Synthetic Minority Oversampling Technique (SMOTE) [16], a commonly used dataset imbalance mitigation technique was adapted. As depicted in Table IV detection with SMOTE yielded a F-measure of 0.958. Also, evaluation on a smaller balanced dataset formed by 500 ad libraries and 500 randomly selected non ad library modules from 15 different categories yielded a F-measure of 0.9414. The aforementioned detection results reveal that AdDetect's classification process is unaffected by the imbalance in dataset.

VI. Discussion

A. Robustness of features

Many ad libraries and apps use obfuscation techniques to circumvent damages caused by easy reverse engineering of apps. With obfuscations tools such as *Proguard* used widely in practice [3], we intend to verify the robustness of features used in AdDetect against common obfuscation techniques. 26 ad libraries in our dataset (including popular ones such as *admob, airpush, youmi and domob*) are found to use obfuscation. AdDetect detects all instances of 25 of these ad libraries, reflecting the framework's resilience to common obfuscation techniques. Few instances of *domob* ad library could not be detected, as explained below.

B. Observations on behavior of ad libraries

Since semantic analysis of ad libraries is maintained at the core of the detection process, we were able to make several key observations, as summarized below.

- Aggressive push ad libraries such as *airpush and sendroid* are found to use services (processes that run in the background) and Android APIs from classes such as *android.app.NotificationManager* to display ads in the devices' notification bar. These characteristics pertain only to push ad libraries, which is not noticed in the case of other ad libraries in our dataset.
- Some ad libraries use client libraries such as *urbanairship* (which facilitates sending push notifications) and *ormma* (which helps in designing rich media content in ads). AdDetect clusters these self contained client libraries as separate modules and correctly detects ad libraries that make use of them.
- Recent versions of ad libraries such as *domob, apperhand, mobclix, vpon,* and *cauly* are found to use native

code which is less common among other ad libraries. Out of these, AdDetect fails to detect some instances of *domob* and *apperhand*, introducing a small fraction of false negatives. *Apperhand* reportedly leaks private information, besides being able to push icons to mobile desktop and bookmarks to browsers. Some security solution providers identify *apperhand* as a malware (Android.Counterclank family) or intrusive adware [17].

- Many ad libraries, including popular ones such as *inmobi* and *airpush*, do not restrict themselves to using only the permissions mentioned in their documentation, but, actively probe for (undocumented) permissions made available through host apps and use them. *smaato* which uses as many as 11 permissions, tops the list of permission greedy ad libraries in our dataset.

VII. Case Study

In this section, the claim that AdDetect complements current research work on analysis of ad libraries is substantiated by conducting a case study on the completeness of a related work, namely, *AdDroid* [5]. *AdDroid*, a study proposing privilege separation between host app and ad libraries, detects and highlights specific cases where ad libraries over privilege the apps. *AdDroid* uses whitelist based approach to detect ad libraries and hence is not guaranteed to find all the ad libraries in an app.

For this case study, a popular app with more than 10 million downloads, namely, Super-Bright LED Flashlight (MD5: 8b2b4b5a41f714a829b495165a9f7ea7) is used. This app includes seven ad libraries: *admob, mobfox, inmobi, millinealmedia, tapjoy, jumptap* and *adfonic*. Out of these only *admob* and *millinealmedia* are in the whitelist maintained by *AdDroid*. Hence, the instances of over privileging due to other five ad libraries could not be detected by *AdDroid*. However, replacing *AdDroid's* whitelist based detection with AdDetect, which is capable of semantically detecting all ad libraries, would help detecting all the instance of over privileging. Table V presents this comparison. It is evident that AdDetect would render completeness to such analysis on ad libraries.

TABLE V
Detection of Over privileging through ad libraries using whitelist based approach vs. AdDetect

Super-Bright LED Flashlight Permissions - over privileged by ad libraries	**Detected by whitelist based approach used in AdDroid[5]**	**Detected by over privileging analysis on top of AdDetect**
ACCESS_FINE_LOCATION	✓	✓
ACCESS_COARSE_LOCATION	✗	✓
ACCESS_NETWORK_STATE	✓	✓
ACCESS_WIFI_STATE	✓	✓
CHANGE_CONFIGURATION	✗	✓
WRITE_SETTINGS	✗	✓

VIII. Conclusion

In this paper, we present, AdDetect, a framework to perform automated semantic detection and recovery of ad libraries from Android apps. An improved module decoupling technique based on hierarchical clustering is proposed to recover individual modules from apps. Semantic features are extracted from these modules and a SVM classifier is built to perform ad library detection. We have implemented a prototype and used it to detect ad libraries in apps from different categories obtained from the official market. Our results show that AdDetect offers 95.34 % accuracy in detecting ad libraries and is robust against obfuscation techniques adopted by ad libraries. We also show that automated semantic ad library detection proposed in AdDetect could complement and render completeness to current research and analysis on ad libraries.

References

[1] Google Play. Dec. 2013. URL: https://play.google.com/store

[2] Grace, M. C., Zhou, W., Jiang, X., & Sadeghi, A. R. (2012, April). Unsafe exposure analysis of mobile in-app advertisements. In Proceedings of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks (pp. 101-112). ACM.

[3] Stevens, R., Gibler, C., Crussell, J., Erickson, J., & Chen, H. (2012, May). Investigating user privacy in android ad libraries. In Workshop on Mobile Security Technologies (MoST).

[4] Book, T., Pridgen, A., & Wallach, D. S. (2013). Longitudinal Analysis of Android Ad Library Permissions. In IEEE Mobile Security Technologies (MoST).

[5] Pearce, P., Felt, A. P., Nunez, G., & Wagner, D. (2012, May). Addroid: Privilege separation for applications and advertisers in android. In Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security (pp. 71-72). ACM.

[6] Tongaonkar, A., Dai, S., Nucci, A., & Song, D. (2013, January). Understanding mobile app usage patterns using in-app advertisements. In Passive and Active Measurement (pp. 63-72). Springer Berlin Heidelberg.

[7] Book, T., & Wallach, D. S. (2013, November). A case of collusion: a study of the interface between ad libraries and their apps. In Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices (pp. 79-86). ACM.

[8] Gibler, C., Stevens, R., Crussell, J., Chen, H., Zang, H., & Choi, H. (2013). AdRob: Examining the Landscape and Impact of Android Application Plagiarism. In Proceedings of 11th International Conference on Mobile Systems, Applications and Services.

[9] apktool. Dec. 2013. URL: https://code.google.com/p/android-apktool

[10] baksmali disassembler. Dec. 2013. URL: https://code.google.com/p/smali

[11] Androguard. Dec. 2013. URL: https://code.google.com/p/androguard

[12] Zhou, W., Zhou, Y., Grace, M., Jiang, X., & Zou, S. (2013, February). Fast, scalable detection of Piggybacked mobile applications. In Proceedings of the third ACM conference on Data and application security and privacy (pp. 185-196). ACM.

[13] Kaufman, L., & Rousseeuw, P. J. (2009). Finding groups in data: an introduction to cluster analysis (Vol. 344). Wiley.com.

[14] Au, K. W. Y., Zhou, Y. F., Huang, Z., & Lie, D. (2012, October). Pscout: analyzing the android permission specification. In Proceedings of the 2012 ACM conference on Computer and communications security (pp. 217-228). ACM.

[15] Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. ACM SIGKDD Explorations Newsletter 11, No 1, pp. 10-18.

[16] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. (2002). SMOTE: Synthetic Minority Over-sampling Technique. Journal of Artificial Intelligence Research, 16: 321-357.

[17] Apperhand ad library threat. Dec. 2013. URL: url-http://www.symantec.com/security_response/writeup.jsp?docid=2012-012709-4046-99&tabid=2