Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

A Permission verification approach for android mobile applications

Dimitris Geneiatakis [a,*], Igor Nai Fovino [a], Ioannis Kounelis [a,b], Paquale Stirparo [a,b]

[a] Institute for the Protection and Security of the Citizen, Joint Research Centre (JRC), European Commission, Ispra, VA, Italy
[b] Royal Institute of Technology (KTH), Stockholm, Sweden

ARTICLE INFO

Article history:
Received 2 January 2014
Received in revised form
6 October 2014
Accepted 14 October 2014
Available online 7 November 2014

Keywords:
Android
Permissions
Security
Instrumentation
Privacy
Risk assessment

ABSTRACT

Mobile applications build part of their security and privacy on a declarative permission model. In this approach mobile applications, to get access to sensitive resources, have to define the corresponding permissions in a manifest. However, mobile applications may request access to permissions that they do not require for their execution (over-privileges) and offer opportunities to malicious software to gain access to otherwise inaccessible resources. In this paper, we investigate on the declarative permissions model on which security and privacy services of Android rely upon. We propose a practical and efficient permission certification technique, in the direction of risk management assessment. We combine both runtime information and static analysis to profile mobile applications and identify if they are over-privileged or follow the least privilege principle. We demonstrate a transparent solution that neither requires modification to the underlying framework, nor access to the applications' original source code. We assess the effectiveness of our approach, using a randomly selected varied set of mobile applications. Results show that our approach can accurately identify whether an application is over-privileged or not, whilst at the same time guaranteeing the need of declaring specific permissions in the manifest.

1. Introduction

Mobile Internet is expected to overtake the usage of land line Internet (Mobithinking, 2013). The reason of this success is not only due to the evolution of smartphones and their underlying infrastructures, but also to the one-stop shop model on which the app-stores (*Google Play*, *Apple store* (*iOS*), *etc.*), are based, enabling the users to purchase the desired application and install it directly on their phones without any additional interventions. These stores, before publishing any application, scrutinize it to identify possible malicious activities by using particular security techniques such as the Google's Bouncer (Android and security, 2012). Though users trust these centralized stores and their security approaches, it is almost impossible to be 100% sure of the secureness of any given application. For instance, Miller and Oberheide (2012) present a technique to bypass *Google's* Bouncer security checks. A similar problem was faced also by *Apple's store* (Ducklin, 2012).

* *Corresponding author.*
E-mail addresses: dimitrios.geneiatakis@jrc.ec.europa.eu (D. Geneiatakis), igor.nai-fovino@jrc.ec.europa.eu (I.N. Fovino), ioannis.kounelis@jrc.ec.europa.eu (I. Kounelis), pasquale.stirparo@jrc.ec.europa.eu (P. Stirparo).
http://dx.doi.org/10.1016/j.cose.2014.10.005
0167-4048/© 2014 Elsevier Ltd. All rights reserved.

These threats acquire a high relevance because today smartphones can be considered *mobile personal inventories*, managing an enormous amount of personal information. This fact, combined with the always online nature of mobile devices makes smartphones a valuable target for attackers. For example, spying applications can collect user's position or steal personal information and sell them to marketing companies (FBI warns loozfon, 2012). Even well-known applications may take advantage of their access to "sensitive" resources for manipulating otherwise personal information as shown in various research works (Stirparo and Kounelis, 2012; Enck et al., 2010; Gibler et al., 2012).

In other cases a mobile application might be *over-privileged*; meaning that it requests more permissions than what it actually needs to accomplish its task. As a result, these applications might be requested by malicious applications to act on behalf of them (Orthacker et al., 2012) and provide access to otherwise private information. In that direction, an adversary could build an over-privileged legitimate application with carefully selected set of "needless" permissions that will transform the application into a malware as soon as an update on the operating system occurs (Xing et al., 2014). Further, end-users might try to install applications from third party stores, which do not scrutinize the functionality of the provided applications at all.

These facts show that the presence of security analysis mechanisms at the store side do not guarantee the security (*e.g.,* lack of malicious operations) and the privacy of end-users personal data. To identify possible mis-configurations and over-privileges in mobile applications researchers focus on different approaches such as:

- Static analysis: Either the source code or the binary of an application are analyzed to identify possible sources and sinks of data leakages (*e.g.,* (Bartel et al., 2012a; Felt et al., 2011; Rosen et al., 2013)) without executing it.
- Dynamic monitoring: The behavior of an application is examined at runtime (*e.g.,* (Enck et al., 2010; Berthome et al., 2012)).
- Scanning applications: Third party applications, like *Permission Explorer* (Permission explorer), are able to scan all the installed applications and generate a user friendly report notifying users for the usage of the requested permissions.
- Operating systems privileges enforcement: Operating systems enforce specific mechanisms in order to eliminate personal data manipulation. For instance, Android OS requests the user to give explicit authorization access to application's specified resources during installation procedure, otherwise the installation fails.

Although these approaches can either identify over-privileged applications or eliminate the chances of manipulating personal data, we believe that an orthogonal approach is required in order to identify and validate the outcomes of such techniques. Static analysis techniques *e.g.,* (Bartel et al., 2012a; Felt et al., 2011; Rosen et al., 2013), usually do not take into account the runtime context, making them prone mainly to false positive identifications or requiring, to be effective, access to applications' source code or/and modification to the underlying framework. For instance, Felt et al. (2011) modify the Android framework to log the permission checks, while solutions such as TaintDroid (Enck et al., 2010) do not focus on identifying over-privileged applications. Furthermore, a service that guarantees the least privilege principle for any given mobile application with a specific degree of certainty is lacking from the current security approaches.

In this work, we elaborate on identifying over-privileges, and validating the need of declaring specific permissions in the manifest of any given Android application, in the context of least privilege principle, by combining static analysis and runtime information. With such an approach, we rely on the advantages of exhaustive static analysis with the discrimination power of dynamic analysis, to provide to the end-user a useful instrument to understand the potential risks associated to mobile applications of uncertain provenance. To the best of our knowledge, this is the first work that proposes a risk assessment framework evaluating application's attack surface exploitation probability due to the permissions declared in the application's manifest.

We focus on the Android OS because it is among the most used operating systems in the market, and it is considered a main target for attackers (Kaspersky security bulletin, 2012). We capitalize on the advantages of Dexpler (Bartel et al., 2012b) and Soot (Vallée-Rai et al., 1999) frameworks to reverse engineer and analyze any given Android mobile application both statically and dynamically. We record and instrument all the possible Application Programming Interfaces (APIs) identified in the reversed engineered code in order to monitor the APIs executed at runtime. Relying on the extracted information we audit the permissions included in the application's manifest and classify application's exposure to over-privileged flaws with certain confidence level. This approach complements other solutions, such as (Bartel et al., 2012a; Felt et al., 2011; Rosen et al., 2013; Berthome et al., 2012), and can accurately *justify* whether or not the application's request to access a specific resource is required. In this way, we also magnify if the application follows the least privilege principle. Results show that our approach can effectively evaluate if any given Android mobile application that provides access to sensitive resources can be manipulated.

The main contributions of this work can be summarized in the following:

- We present a risk assessment framework with respect to permission requesting access to sensitive resources. The proposed framework automatically assures whether an application follows the least privilege principle, and identifies over-privileges with certain confidence level.
- This is the first work on Android mobile application's permissions assessment that (a) links static and runtime information, (b) operates transparently with respect to the OS and the application's original source code, and (c) requires little or no supervision.
- Our solution's software is freely available.[1] We believe this can facilitate experimentation with detecting over-privileged applications.

[1] http://code.google.com/p/android-app-analysis-tool/.

The rest of the paper is structured as follows. In Section 2 we provide an overview of the Android OS permission security model and we describe the security issues introducing the over-privileged applications in Section 3. In Section 4 we outline our framework for identifying over-privileged applications by combining static and dynamic analysis information, while in Section 5 we evaluate the proposed framework in terms of effectiveness. We comment on the findings of our approach in Section 6 and we overview other similar works in Section 7. In Section 8 we illustrate the limitations of our approach and present some pointers for future work. Finally, in Section 9 we draw our conclusions.

2. Android permission-based security model

The core of the Android OS is built on top of the Linux kernel. This enables it to provide strong isolation for protecting users data, system resources and avoiding conflicts, for both Java programming language and native Android mobile applications. Fig. 1 overviews the Android OS architecture.

The Android OS system runs each application under the privileges of different "user", and assigns a unique user ID to each of them. This approach differs from other operating systems where multiple applications run under the same user's permissions. By default, applications are not allowed to execute functions that might affect other applications or users, and they have access to a limited set of resources. Applications must mandatory declare in a manifest (see Listing 1[2]) all the "sensitive" operations that can take place in the course of execution; the users, during the installation, are requested to endorse them, otherwise the installation fails. In case an application executes a protected feature that has not been declared in the manifest, a security exception will throw during execution.

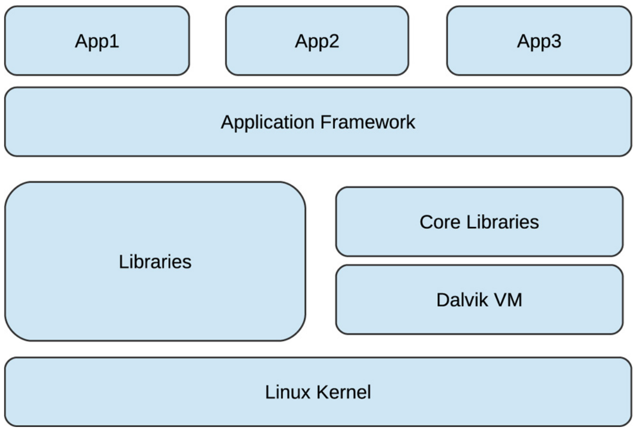


Fig. 1 – Android software modular architecture.

text messages, emails, *etc.*, they generate new opportunities for profiling users' and manipulating these data in return of financial benefit in different ways. Not only malicious applications (malware) misuse such information, but even legitimate ones. For example, the *Twitter mobile application* sent out users' personal information, without their consent (Mobile apps take data without permission, 2012). Users' data can be modified or even lost in case that such applications are allowed to execute sensitive operations *e.g.,* delete, modify, *etc.* A detailed analysis of personal data manipulation in mobile applications can be found in Enck et al. (2010), Gibler et al. (2012), Stirparo and Kounelis (2012).

In other cases, malware might exploit legitimate applications' configuration vulnerabilities, and/or manipulate their permissions to gain access either to private information or to other protected functionalities that should provided only to legitimate applications. This, for instance, can be achieved

```
<android.permission.CAMERA/>
<android.permission.WRITE_EXTERNAL_STORAGE/>
<android.permission.INTERNET/>
<android.permission.ACCESS_NETWORK_STATE />
<android.permission.READ_PHONE_STATE/>
<android.permission.READ_CONTACTS/>
<android.permission.VIBRATE/>
<android.permission.WRITE_CALENDAR/>
```

Listing 1: An example of a real Android mobile application manifest records. The application requests access to various resources such as *Camera, Internet, Calendar, etc.*

3. Threat model: permissions' Back-Doors

As mobile applications manage a wide range of personal information, such as unique identifiers, location, call history,

through inter-process communication as demonstrated in Orthacker et al. (2012), without the need to exploit a vulnerability, neither at the application nor at the OS level. In that direction, application that declared more permissions than those they actually need (over-privileged application) can be transformed silently to malware, whenever an operating system or an application update occurs. In that case, if the needless permissions do not exist in the operating system where the application will be installed they will be ignored. These permissions will be silently granted as soon as an update occurs driving to privilege escalation through updating (pileup) (Xing et al., 2014), without users' consent.

[2] The proper syntax is the following: uses-permission android: name = permission-name.

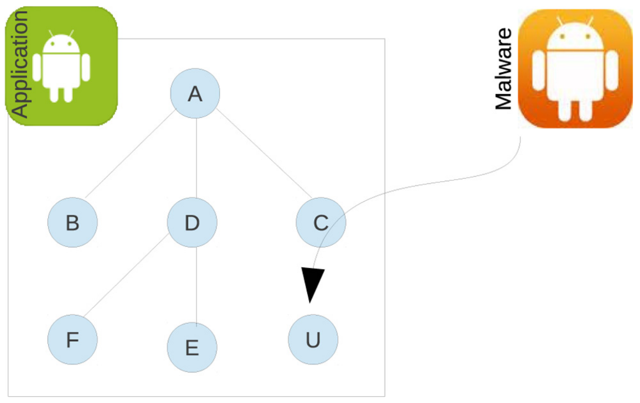


Fig. 2 – Example of exploiting vulnerabilities, by a malware, on the method *C* in order to execute the method *U* that is "out of the scope" of the application. By the term "out of the scope" we mean that the malware is taking the advantage of the application's needless privileges to execute the method *U*.

In another approach, the attacker might request access to a set of permission explicitly during the installation pretending the legitimacy of the installed application. In that case, a malware might exploit a specific vulnerability that will allow the execution of an API that otherwise would not be possible to execute, as illustrated in Fig. 2. *Webview*, for example, is vulnerable to malicious input, as referred in Stephan and Siegfried (2013). Consequently, the malware, in that instance, can execute any API, if the exploited application has the appropriate permissions.

Attackers might follow such approaches in an attempt to hide their malicious activity and evade any intrusion detection service. These security flaws come into existence mainly due to the fact that the Android OS assumes that an application's permission restricted functionality, will be properly used by applications. Moreover, it should be noticed that such threats do not focus on software vulnerabilities in OS or in the application layer. Instead, they exploit misconfiguration at the application layer that currently users are not in the position to validate. As a result malicious applications will try to benefit from such kind of security weaknesses to access sensitive services and data without being noticed by the end user, in an architecture where the least privilege principle is believed to be strictly adopted. Thus, we defend that, to enhance the security and privacy level of the end-user, the identification of over-privileged applications is of high importance.

4. Proposed approach

As described in the introduction, we are interested in defining a method allowing to effectively profile and analyze mobile applications, in search for over-privileges. The approach adopted is based on application repackaging to verify the real need of requesting and granting access to all the "sensitive" Android's APIs.[3] This approach requires neither access to the applications' original source code nor modification of the underlying framework. We rely (a) on static analysis to compute the (maximum) set of permissions that might be used by the examined application, and (b) on dynamic analysis to validate their use. The outcomes of both static and dynamic analysis are combined and compared with the manifest's permission set to deduce whether the application is over-privileged or not.

[3] Sensitive APIs, as defined by Android OS, are the ones that need to be declared in the manifest.

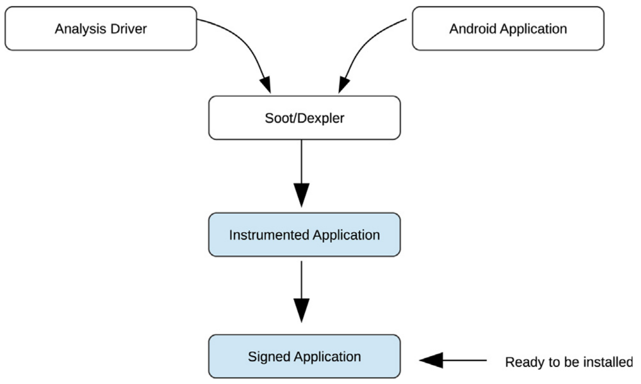


Fig. 3 – A high level procedure for repackaging Android mobile applications.

4.1. *Application analysis and repackaging*

Android mobile applications target the Dalvik, instead of pure Java, bytecode. Java based applications can be reverse-engineered by using tools such as Jad (Jad java decompiler) and ASM (ASM). Similarly, the android mobile applications can be analyzed and modified by using tools such as ApkTool (A tool for reverse engineering android apk files), Androguard (Androguard), Dexpler (Bartel et al., 2012b) and Dex2Jar (Dex2Jar). Note that in most cases Dalvik bytecode is not transformed to the original Java code but to an intermediate format, depending on the tool's capabilities. In this work, we rely on Dexpler (Bartel et al., 2012b) a Soot framework (Vallée-Rai et al., 1999) based tool for analyzing, modifying and repackaging android mobile applications. In this framework, any Android mobile application can be given, as input, to a properly designed analysis driver to analyze statically application's reversed engineered code, and introduce additional functionality to it, depending on the requirements.

Consider for instance the case where we are interested to record (*e.g.*, in a file) the calls made towards the `getDeviceId` API at runtime. The mobile application is transformed to Jimple interpretation (Vallee-Rai and Hendren, 1998) through Dexpler, which enables the usage of Soot framework (Vallée-Rai et al., 1999). The analysis driver iterates on the code to identify the method `getDeviceId` in which the monitor code is injected. This task is accomplished by the code illustrated in Listing 2. As soon as the analysis is completed the application can be signed, installed and executed on the Android phone.

Listing 2: An example of Soot framework code to analyze an Android mobile for the identification of a particular method such as getDeviceId.

```
protected void
internalTransform(Body bd,String pNm, Map op)
{
  Chain units = bd.getUnits();
  Iterator stmtIt = units.snapshotIterator();
  while(stmtIt.hasNext())
    Stmt s = (Stmt) stmtIt.next();
    InvokeExpr iexpr = s.getInvokeExpr();
    if(iexpr instanceof InvokeExpr)
      SootMethod trgt = iexpr.getMethod();
      if(trgt.getMethod().equals("getDeviceId()"))
        //monitor code
}
```

The general procedure for analyzing and repackaging an Android mobile application is illustrated in Fig. 3.

4.2. *Theoretical framework*

In this section we present the theoretical framework on which we based our approach.

An application is considered *over-privileged* if and only if there is a permission object in the manifest set (M_p) which is not listed in the static analysis permission set (S_p) (*i.e.*, the set compiled during the static analysis of the application), as illustrated in Equation (1). The over-privilege set is computed as the intersection between the M_p and S_p as illustrated in Equation (2). Complementarily, an application is marked as non over-privileged if and only if the static, dynamic and manifest permission sets match as illustrated in Equation (3). These cases are graphically illustrated in Fig. 4 as well. In the case where the manifest (M_p) and the static (S_p) permission sets are equal, but they are a superset of the dynamic (D_p) permission set (refer to Equation (4)) then a specific set of the examined application's permissions, according to Equation (5), might be susceptible to over-privileged flaws with a certain confidence level.

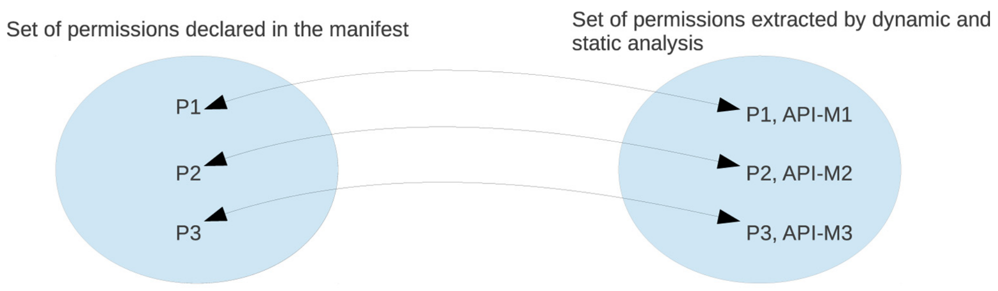
$$S_p \subset M_p \rightarrow Over-privileged \tag{1}$$

$$M_p \backslash S_p \rightarrow Over-privilege-permission-set \tag{2}$$

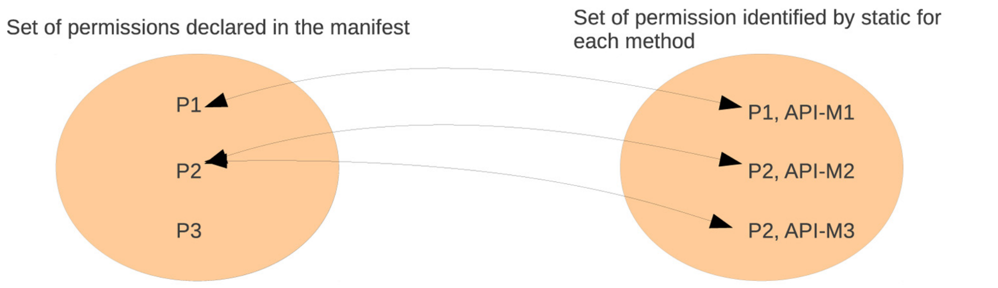
$$M_p \cap S_p \cap D_p = M_p \rightarrow Non-over-privileged \tag{3}$$

$$D_p \subset (M_p = S_p) \rightarrow Susceptible-attacks \tag{4}$$

$$M_p \backslash D_p \rightarrow Permission-set-susceptible-to-attacks \tag{5}$$


Set of permissions declared in the manifest
Set of permissions extracted by dynamic and static analysis
P1
P2
P3
P1, API-M1
P2, API-M2
P3, API-M3

(a) A validated non over-privileged set.


Set of permissions declared in the manifest
Set of permission identified by static for each method
P1
P2
P3
P1, API-M1
P2, API-M2
P2, API-M3

(b) An over-privileged set.

Fig. 4 – Definition of validated non over-privileged and over-privileged application.

In details, we audit whether the S_p is a subset of the M_p or not (refer to Equation (1)); the difference between the M_p and S_p generates the over-privileged set, according to Equation (2). *If this is the case*, then we deduce that the application is over-privileged with probability 1. This is due to the fact that one or more permission objects belonging to M_p are not mapped to any of the permission objects of the S_p identified in the reverse engineered application. *If not*, we examine the D_p and compare it with the M_p and the S_p relying to Equations (3)–(5). In that case, we deduce that the application is not over-privileged with probability of 1 if these sets are equal, since the runtime information matches with static analysis outcome, according to Equation (3). This means that all the sensitive methods, requesting specific permission to be executed, were reached during runtime, and thus (a) the examined application is not susceptible to over-privileged threats, and (b) it is guaranteed that the application respects the least privilege principle.

In the case that the D_p is a subset of the M_p the examined application, according to Equation (4), is susceptible to over-privileged flaws with a certain confidence level. This might also correspond to a dynamic analysis false identification. To mitigate such false identifications due to the limited coverage that, in some situations, might be achieved during the dynamic analysis, the final set of the over-privileges combines the outcomes of the static and dynamic phases according to Equation (6).

$$(M_p \backslash D_p) \cap (M_p \backslash S_p) \rightarrow Final-over-privileged-set \quad (6)$$

Note that if any of the sets computed by Equations (2) and (5) equals to an empty set, then we rely only on the non empty set to compute the final over-privilege set.

4.3. *Permissions' risk & false identification assessment*

To assess possible false identifications that our approach might generate and measure the exposure of the examined application to over-privileged threats we rely on the attack surface introduced by the set of privileges not reached during the dynamic analysis phase (refer to Equation (7)), where S corresponds to the percentage of the permissions triggered during the execution according to the Equation (8).

$$E_a = 1 - S \quad (7)$$

$$S = \frac{|M_p| - |D_p|}{|M_p|} \quad (8)$$

It should be mentioned also that if the static analysis set is a superset of the manifest one, we analyze only the manifest set. This is because Android OS will throw a security exception, according to its security design, for any sensitive API having no declaration of the permissions needed for its execution in the application's manifest. Table 1 summarizes the risk assessment classes provided by the proposed framework.

4.4. *Identify over-privileged applications*

To determine *over-privileged* applications, we integrated the Dexpler (Bartel et al., 2012b) and Soot framework (Vallée-Rai et al., 1999) with an analysis driver that:

1. Identifies and records all the methods existing in the reversed engineered application— Static analysis phase.
2. Injects small pieces of monitoring code before every API call provided by the Android OS and records its name in the private storage area of the analyzed application— Dynamic analysis phase. This allows us to run the application without the need to modify the manifest of the original application.

The output of the analysis extracts the manifest's permission set (M_p) and generates a new Android mobile application package, the instrumented one, which records (a) all the possible methods that exist in the reversed engineered application, and (b) the methods executed at runtime. Note that the reversed engineered code does not necessarily correspond to the application's original source code, however, the API calls will be the same of the original code.

The instrumented application can be executed either manually or automatically relying on the Android monkey test suite,[4] which is able to generate and inject to the examined application pseudo-random streams of user events (*e.g.*, clicks, touches,*etc.*) in a random but repeatable test cases. The Monkey test suite is an official Android framework that can be used by developers for checking application's robustness before publishing it. This way, we reproduce common real life situations and record the executed methods. In case that additional inputs are available, *e.g.*, when users' exercise the application, they can be injected through the virtual device drivers available in Android. When the execution is completed, we create the methods (APIs) permission map for both the static and the dynamic analysis, and we determine:

1. The set of permissions included in the reversed engineered application, correspond to the static analysis permission set (S_p).
2. The permission set required for the examined applications execution, correspond to the dynamic analysis set (D_p).

This is achieved by identifying the signature of the executed call in the permission mapping database, based on the permission mapping published in Felt et al. (2011). Afterward, we compare the permissions sets identified in the previous step with those included in the manifest to deduce whether or not the examined application is over-privileged. The whole procedure to identify an over-privileged application is illustrated in Fig. 5. As soon as the sets M_p, S_p and D_p have been produced we assess applications' permissions exposedness based on the theoretical framework described in Section 4.2.

Table 1 – Android mobile applications privileges risk assessment.

Classification-id	Assessment	Exposure probability
(1)	Over-privileged	1
(2)	Over-privileged	Refer to Equation (7)
(3)	Non over-privileged	0

[4] http://developer.android.com/tools/help/monkey.html.

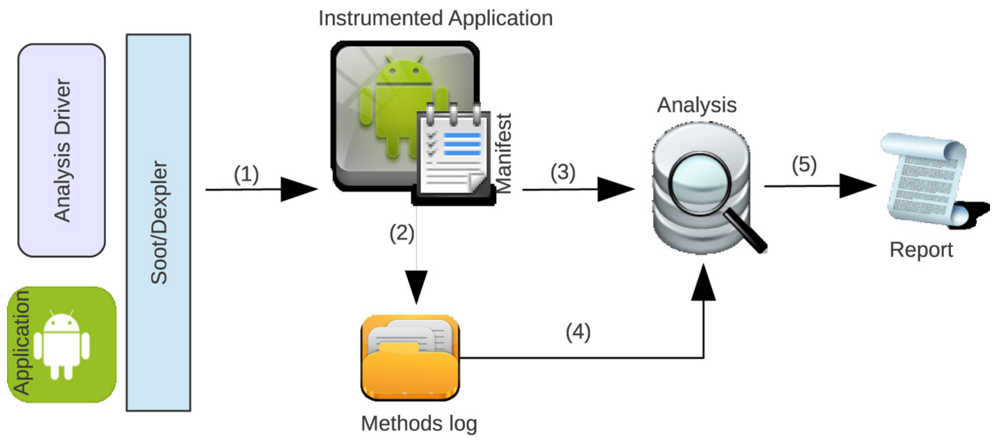


Fig. 5 – Proposed permission verification approach. The soot framework analyzes the mobile application and inserts small pieces of code to monitor the executed methods (1). Every time the application is executed all the methods are logged (2) and analyzed (3,4) in order to determine whether the application uses all the requested permissions (5), according to the proposed theoretical framework.

5. Evaluation

We analyzed 265 Android mobile applications, both on real device and on Android emulator running Android 4.2.2 version, to demonstrate and evaluate the effectiveness of our approach in (a) classifying applications as over-privileged or not, and (b) assess the probability to be exploited due to flaws introduced by over-privileges. The examined applications were randomly selected, belonging to the top 500 in their categories on the Google Play, and were distinguished based on their functionalities as follows:

1. Expenses: Manage users' financial transactions.
2. Linguistic: Provide language tests.
3. Shopping: Manage daily shopping needs.
4. Entertainment: Applications for entertainment such as games.
5. Accessories: Support users' in various daily tasks (*e.g.*, notes, bookmarks, *etc.*).
6. Hello: A reference application developed by us, which shows a hello-world message and writes it in the external secure digital (SD) storage.

We exercise applications automatically based on the Android's Monkey test suite, by injecting 1500 different events, to extract the runtime information, which includes the called methods, while the static analysis information was generated during instrumentation phase (see Fig. 5).

The procedure for generating the S_p and D_p sets during the evaluation is illustrated in Fig. 6. Results, as summarized in Fig. 7, show that the majority of the examined application (87%), are over-privileged by design (classification category—1). This practice could be explained with the attitude of developers to ask for the maximum number of permissions during the first installation, to avoid to request for extra permissions when the application is updated.

Another 10% belong to over-privileged (classification category—2), while only 3% of the examined applications were validated as non over-privilege (classification category— 3).[5] These tests clearly illustrate the feasibility of our approach. Obviously, the analysis of 265 applications cannot be considered a statistical population from which derive a general trend, but indeed this is not the scope of this work.

In more detail, Table 2 summarizes and represents indicative outcomes[6] of our analysis with regard to the effectiveness of our approach to identify whether or not a given application is over-privileged during our testing campaign. The most common needless permissions the over-privileged applications request are illustrated in Table 3, while Table 4 indicates the over-privileged permissions granted to the same set of applications as identified in the different phases of our approach. For example, the examined over-privileged application Shopping (3) requests access to permissions such as the READ_SMS and the READ_CONTACTS. However, in reality there was not any direct functionality using these permissions. Consequently, malware might exploit these permissions to gain access to otherwise private information.

6. Discussion

To the best of our knowledge this is the very first work that investigates the possibilities of validating the usage/need of declaring specific permissions in the manifest of Android mobile applications, by combining static and runtime information, in the direction of a risk assessment approach. We remind here that the Android OS uses the permissions as a mechanism to protect access to "sensitive" APIs—resources. In this way, if an application follows the least privilege principle (Bishop, dec 2002) a potential exploitation would have a minimum impact. However, as mentioned previously, the

[5] Recall that the different classes of our risk assessment approach are summarized in Table 1.
[6] We do not explicit refer to the names of the examined applications, instead we encoded their names as follows: Category (application number).

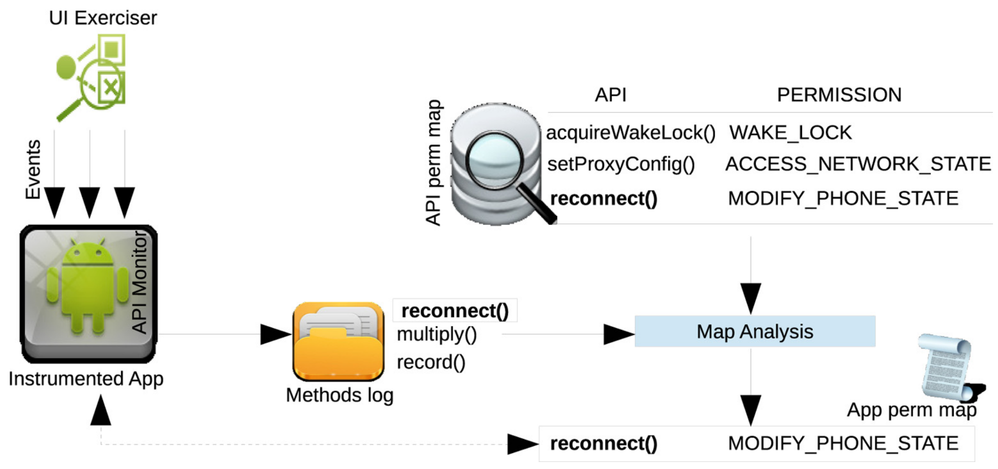


Fig. 6 – APIs—Permissions map example generation. The same is the procedure of both static and dynamic analysis.

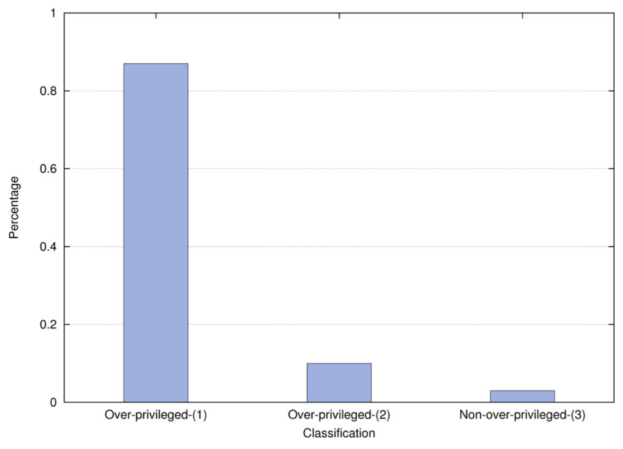


Fig. 7 – Our approach classification distribution for 265 examined mobile applications.

Table 3 – Most used permissions among over-privileged applications as identified by our approach.

Permission type	Usage by application
WRITE_EXTERNAL_STORAGE	11%
RECEIVE_BOOT_COMPLETED	9%
READ_PHONE_STATE	9%
ACCESS_NETWORK_STATE	18%
ACCESS_COARSE_LOCATION	11%

existence of needless permissions offers the chances to bypass this protection mechanism. Malicious application might leverage on this source of potential threats to access either personal data or some other "sensitive" functionality, as demonstrated for instance in Xing et al. (2014). The outcomes of our tests reveal that it is quite common to have applications requesting access to permissions that are not indeed required for carrying out their tasks.

With regards to the accuracy with which this approach is able to classify an application as over-privileged or not, we rely on legacy Intrusion Detection Systems error assessment metrics, namely False Positive (FP) and False Negative (FN) (Gu et al., 2006). The first one is related to applications classified as over-privileged, but in reality they are normal applications, while the latter involves applications classified as normal, but belonging instead to the set of over-privileged applications'.

Table 2 – Indicative assessment outcomes for a sample of different types of applications achieved by the proposed solution. We identify both the number of the methods that exist in the reversed engineered applications code and those executed during the evaluation. The examined applications are classified to the corresponding over-privilege class.

Type	Methods	Executed methods avg.	Over-privileged (classification)	Exposure probability
Expenses(1)	1830	558	Yes (1)	1
Entertainment(1)	1873	31	No (3)	–
Accessories(1)	559	113	Yes (1)	1
Expenses(2)	3910	925	Yes (1)	1
Linguistic	2105	605	No (3)	–
Shopping(1)	1848	596	Yes (1)	1
Hello	2113	18	No (3)	–
Shopping(2)	505	136	No (3)	–
Shopping(3)	257	125	Yes(1)	1
Shopping(4)	211	132	No (3)	–
Entertainment(2)	2180	74	Yes (1)	1
Accessories(2)	396	138	No (3)	–
Accessories(3)	718	11	Yes (2)	0.66

Table 4 – Over-privilege list as identified by our approach in all the stages.

Application type	Static set—Equation (3)	Dynamic Set—Equation (5)	Final set—Equation (6)
Expenses(1)	READ_EXTERNAL_STORAGE ACCESS_COARSE_LOCATION	READ_EXTERNAL_STORAGE ACCESS_COARSE_LOCATION READ_PHONE_STATE	READ_EXTERNAL_STORAGE ACCESS_COARSE_LOCATION
Entertainment(1)	–	–	–
Accessories(1)	WRITE_EXTERNAL_STORAGE WRITE_HISTORY_BOOKMARKS	WRITE_EXTERNAL_STORAGE WRITE_HISTORY_BOOKMARKS	WRITE_EXTERNAL_STORAGE WRITE_HISTORY_BOOKMARKS
Expenses(2)	CAMERA READ_CALENDAR RECEIVE_BOOT_COMPLETED WRITE_CALENDAR	CAMERA READ_CALENDAR RECEIVE_BOOT_COMPLETED WRITE_CALENDAR WRITE_EXTERNAL_STORAGE READ_CONTACTS VIBRATE	CAMERA READ_CALENDAR RECEIVE_BOOT_COMPLETED WRITE_CALENDAR
Shopping(1)	READ_CONTACTS READ_PHONE_STATE RECEIVE_SMS SEND_SMS	READ_CONTACTS READ_PHONE_STATE RECEIVE_SMS SEND_SMS	READ_CONTACTS READ_PHONE_STATE RECEIVE_SMS SEND_SMS
Linguistic	–	–	–
Hello-World	–	–	–
Shopping(2)	–	–	–
Shopping(3)	READ_CONTACTS READ_SMS	READ_CONTACTS READ_SMS	READ_CONTACTS READ_SMS
Shopping(4)	–	–	–
Entertainment(2)	– –	ACCESS_NETWORK_STATE INTERNET READ_PHONE_STATE	INTERNET READ_PHONE_STATE
Accessories(2)	–	–	–
Accessories(3)	–	ACCESS_NETWORK_STATE VIBRATE	ACCESS_NETWORK_STATE VIBRATE

Since we do not have the original source code of the examined applications we cannot have an accurate indication of the applications code covered during dynamic analysis. This is because Java application's reverse engineered "source code" consists of thousands of reachable methods, even for the single Java *Hello-World* application (Ali and Lhoták, 2012). This is also the case for the Android mobile applications relying on Java as indicated in our results (refer to Table 2). Thus, we assess the false identification rate in terms of permissions not *triggered* during the execution, as described in Section 4.

Applications classified in category—1 are vulnerable to over-privileged flaws with probability 1—meaning no false identifications. This is due to the fact that the manifest contains permissions that are not mapped to any API in the examined application' reverse-engineered code. Similarly, the classification of not over-privileged applications (category—3) do not include any false identification, since the manifest's permissions of the examined application were matched not only with those determined in the static analysis phase, but also with those of the dynamic one. Finally, in the cases where the applications were classified as over-privileged in category—2, we assess the false positives in terms of the permissions' not reached during the dynamic analysis. That is equivalent to the exposure probability. For instance, in case of the application we named Accessories(3) in our tables, the false positive assessment is 66% because during the execution we did not reach 2 of the 3 permissions declared in the manifest. This is also the case for the permissions false positive identification themselves. However, if the map of *APIs—Permission* in which we rely on is incomplete, then the proposed approach will generate a false positive.

With this approach we combine the advantages of static and dynamic analysis in order to eliminate false identifications, both for the application and the permission classification itself. On the one side, the dynamic analysis part guarantees the correctness of the non over-privileged application— meaning that applications follow the least privilege principle. Moreover, the dynamic analysis part offers the ability to assess possible mis-classifications in terms of FN that the static analysis module might generate if it is employed by itself. The FN equals to the exploitation probability in terms of non-reachable permissions, according to Equation (7). Such cases are generated when $M_p=S_p$ but actually parts of the permissions target a new version of the OS, as demonstrated in the pileup flaw (Xing et al., 2014). On the other side, the static analysis magnifies all the possible permissions and guarantees the correctness of the over-privileged identification classified in category—1. This means that static and dynamic analysis modules complement each other.

Moreover, relying on the proposed approach, the false positive alarms generated by other static analysis approaches can be eliminated. For instance, in Stowaway (Felt et al., 2011), a solution that relies on static analysis, authors mention that their approach generates false positives.[7] This is due to the fact that Stowaway does not take into account which parts of

[7] The discussion of false negatives in Stowaway is considered as a part of Stowaway authors' future work.

the code are executed. We encountered such a case when we used Stowaway to analyze the *Hello* application. In details, Stowaway identified the permission WRITE_EXTERNAL_STORAGE as unnecessary, and characterized this application as over-privileged. However, the application needed this permission to execute a write operation in the external storage. We are aware of this since we developed the *Hello* application in order to use it, among the others, as a demonstrator of the proposed approach. Stowaway, also, assumes the need of the WRITE_EXTERNAL_STORAGE permission if they identify an API call that returns a path to the SD card directory such as `Environment.getExternalStorageDirectory()`. However, this does not seem to be the case, since we use this particular API in our demo *Hello* application, while the Stowaway online analysis tool considers the WRITE_EXTERNAL_STORAGE permission as an extra permission.

Table 5 overviews the outcomes provided by Stowaway, for the same sample set of applications we examined with our framework, and presented as indicative outcomes in this work. As results Stowaway is less accurate in terms of determining over-privileges. We compared explicitly the outcomes of our approach only with the Stowaway solution as no other solution provides the code or an online service for analyzing applications' permissions.

As additional consideration, one might argue that permission scanning applications can determine which permissions are required in order to execute an application. However, such applications simply read the manifest of a given application without carry out any type of analysis on it. Consequently, they neither provide any valuable information on how the examined applications' declared permissions are used, nor deduce whether an application is over-privileged. We should also note that if an application is not over-privileged does not necessary mean that it is not a malware and vice versa. An application can be infected by a malware either it is over-privileged or not; it may be the case that a pure malware application does not over use privileges. Therefore, our findings do not directly point out malware applications but reveal (a) poor programming techniques from the developers side, and (b) potential points of manipulation.

Table 5 – Over-privilege list as identified by Stowaway (Felt et al., 2011).

Type	Over-privilege set
Expenses(1)	READ_EXTERNAL_STORAGE ACCESS_COARSE_LOCATION
Entertainment(1)	–
Accessories(1)	–
Expenses(2)	WRITE_EXTERNAL_STORAGE
Shopping(1)	–
Linguistic	–
Hello-World	WRITE_EXTERNAL_STORAGE
Shopping(2)	–
Shopping(3)	READ_SMS
Shopping(4)	–
Entertainment(2)	INTERNET READ_PHONE_STATE
Accessories(2)	–

7. Related work

In this section we mainly overview the works focus on the elimination of users' privacy violations and permission manipulation for Android and iOS operating systems as they are the most used in the market. To clarify the reading of the section, we clustered the solutions in four main groups.

Operating system and other general solutions: To eliminate the risk of personal data manipulation Android and iOS operating systems follow different approaches. On the one hand, Android OS (Google) provides strong application isolation. By default applications are not allowed to execute functions that affect other applications or the user. Applications have to declare in a manifest all "sensitive" operations that can be accomplished during their execution, which the user should endorse during installation. Android does not offer any capability to users for dynamically enabling permissions. We should also note that solutions provided by phone manufacturers such as KNOX,[8] though enhance users' authentication, secure communication channels and can control users' data to eliminate possible data leaks, does not focus on the identification of over-privileged applications. On the other hand, iOS (Apple) since version five, did not incorporate any functionality to avoid data manipulation; iOS in fact, protects users' data through developer license agreement. In the latest release iOS enables users to enhance the control of their personal data by requiring applications to get explicit permission before accessing them. However, not only the underlying security mechanism can be by-passed (*e.g.*, (Miller and Oberheide, 2012; Apple)), but, even worst, "benevolent" applications might manipulate (a) personal data as demonstrated in Enck et al. (2010), Gibler et al. (2012), or/and (b) the granted permissions (Xing et al., 2014). In this context, the security level of the Android OS is criticized in Shabtai et al. (2010). Thus, various researches have been published to enhance the security and privacy levels in the mobile platforms relying either on dynamic or static analysis of the application or/and the underlying framework—operating system.

Dynamic analysis security and privacy solutions: TaintDroid (Enck et al., 2010) describes an extension to the Android platform that tracks the flow of sensitive data through third-party applications in order to identify possible data leaks. In the same direction, AppsPlayground (Rastogi et al., 2013) introduces a framework to enable dynamic security analysis with no supervision for detecting privacy leaks and malicious functionalities. Similarly, solutions such as (Schreckling et al., 2013; Kodeswaran et al., 2012) deploy a runtime monitor for enabling users to control their data through their defined policies. Feth and Pretschner (2012) introduce on the TaintDroid (Enck et al., 2010) the notion of fine grained security policies to monitor applications' behavior, while Mockdroid (Beresford et al., 2011) allows users to revoke access to particular resources at run-time. Analogous research works have been accomplished for iOS (Werthmann et al., 2013; Egele et al., February 2011). To avoid the modifications in the

[8] http://www.samsung.com/global/business/mobile/platform/mobile-platform/knox/index_devicesecurity.html.

underlying framework (*e.g.,* middleware, OS,*etc.*) Berthome et al. (2012) propose application's repackaging approach in which the compiled application is analyzed and injected with particular code at the bytecode level for monitoring all accesses of personal data. DroidScope (Yan and Yin, 2012) relies also on dynamic analysis, and introduces a fine-grained instrumentation framework, for analyzing applications which are known to be malware to generate attack signatures. Other virtualization platforms can be used to build a solution for detecting personal data manipulation such as (Barr et al., Dec. 2010), malicious software, and over-privileged applications, with approaches similar to Argos (Portokalidis et al., 2006), however, this domain of research is out of the scope of this work. To the best of our knowledge, none of the existing dynamic analysis approaches are used in the realm of over-privileged applications.

Static analysis security and privacy solutions: In Xiao et al. (2012), Gibler et al. (2012) are introduced complementary approaches to dynamic ones based on static analysis to classify the information flows inside the application as safe or unsafe in terms of privacy. AppProfiler (Rosen et al., 2013) develops a knowledge base of privacy related behaviors, which is used to assess the privacy "level" of a given application. ScanDroid (Fuchs et al., 2009) extracts security specifications from the manifest of the examined application and checks through static analysis whether data flows is consistent with this specifications, however, this solution has not yet been tested in real-world applications.

Privilege based solutions: Besides the techniques used to eliminate the privacy violations by controlling users data, other works focus on privilege usage. Bartel et al. (2012a) and Felt et al. (2011) introduce two different solutions, which rely on application static analysis, to identify over-privileges for any given application. Each of these solutions develop a permission map first, and then rely on static analysis to identify possible permission manipulation. Xing et al. (2014) introduce a scanner tool for detecting applications that are vulnerable against the pileup threat. In this approach, the released Android OS images are analyzed and retrofit a mobile application, named SecUP, installed on users' phones which extracts applications' manifest information in order to identify the existence of vulnerable applications. This is a similar but alternative solution to our approach.

Barrera et al. (2010) introduce a methodology based on self-organized maps for assessing Anrdoid's permission model and investigate how permissions are used by applications. Android's OS permission evolution over time is studied in Au et al. (2012), Wei et al. (2012). To do so, PScout (Au et al., 2012) develops a static analysis tool to analyze Android's OS and extract its permission related features, while Wei et al. (2012) focus mainly on the "developments" of needless permissions based on the Stowaway solution. In an alternative approach Felt et al. (2012) accomplish a thorough survey to examine the effectiveness of the permission system in terms of supporting users to take the appropriate security decisions for permission granting based on their needs. Whyper (Pandita et al., 2013) introduces a natural language processing technique that enables the deduction of requesting access to specific permission based on applications' description. Stevens et al. (2013) propose a statistical model for predicting permission misuse by relating the use of common permissions with questions appeared in Stack-Overflow website.

Table 6 summarizes the different features supported by existing solutions. Approaches focusing on over-privileges detection, mainly rely on static analysis, without taking into account the real time context. We believe that our approach complements other solutions such as Bartel et al. (2012a), Felt et al. (2011), and improves detection accuracy.

Table 6 – Features supported by different solutions.

Solution	Static	Dynamic	Privacy	Malware	Over privilege	Permission evolution	Other
TaintDroid (Enck et al., 2010)	–	✓	✓	–	–	–	–
AndroidLeaks (Gibler et al., 2012)	✓	–	✓	–	–	–	–
Bartel et al. (Bartel et al., 2012a)	✓	–	–	–	✓	–	–
Felt et al. (Felt et al., 2011)	✓	–	–	–	✓	–	–
AppProfiler (Rosen et al., 2013)	✓	–	✓	–	–	–	–
Berthome et al. (Berthome et al., 2012)	–	✓	✓	–	–	–	–
AppsPlayground (Rastogi et al., 2013)	–	✓	✓	✓	–	–	–
Kynoid (Schreckling et al., 2013)	–	✓	✓	–	–	–	–
Kodeswaran et al. (Kodeswaran et al., 2012)	–	✓	✓	–	–	–	–
Feth et al. (Feth and Pretschner, 2012)	–	✓	✓	–	–	–	
Mockdroid (Beresford et al., 2011)	–	–	–	–	–	–	✓
PSiOS (Werthmann et al., 2013)	–	–	–	–	–	–	✓
PiOS (Egele et al., February 2011)	–	–	–	–	–	–	✓
DroidScope (Yan and Yin, 2012)	–	✓	–	✓	–	–	–
Xiao et al. (Xiao et al., 2012)	✓	–	✓	–	–	–	–
SCanDroid (Fuchs et al., 2009)	✓	–	✓	–	–	–	–
Barrera et al. (Barrera et al., 2010)	–	–	–	–	–	✓	–
PScout (Au et al., 2012)	–	–	–	–	–	✓	–
Xuetao et al. (Wei et al., 2012)	–	–	–	–	–	✓	–
Felt et al. (Felt et al., 2012)	–	–	–	–	–	✓	–
Whyper (Pandita et al., 2013)	–	–	–	–	–	–	✓
Ryan et al. (Stevens et al., 2013)	–	–	–	–	–	–	✓
Our solution	✓	✓	–	–	✓	–	–

8. Limitations and future work

Our approach's main limitation lies in the fact that the dynamic analysis might generate false positive alarms, since we cannot guarantee a complete code coverage for all the sensitive APIs identified in the reverse engineered code of the examined application. Even if currently we rely on a general test suite to execute and navigate automatically the instrumented application, there will be code paths that might not be covered. To maximize the code coverage we are planing to introduce in a future work (a) a capture-replay approach for simulating real users' interaction, which however, needs first manual execution, and (b) symbolic execution which appears to be a good option for state space exploration of an application.

Moreover, we are considering to exploit our solution to study the over-privilege phenomenon on large scale. However, because of application repackaging, not all the applications can be executed successfully as the generated code might violate the Android OS execution environment. This is a current limitation of Dexpler (Bartel et al., 2012b) as it does not handle optimized Dalvik (odex) opcodes. In addition, when Dexpler infers types for ambiguous declarations the algorithm supposes that the Dalvik bytecode is correct, which might not be the case under all circumstances. Currently, we are looking on these cases in order to eliminate such problems.

9. Conclusions

Over-privileged applications introduce new possibilities for manipulating personal information and sensitive functionalities managed in smart-phones. Users have to trust applications requests for accessing sensitive resources, defined in their manifests, if they would like to install and use them. However, even benevolent over-privileged applications might be exploited by malicious applications to gain access to otherwise not-accessible (personal) data. In this paper, we introduce an Android mobile application permissions' risk evaluation approach that combines static and dynamic analysis to assess any given application as over-privileged with certain degree of probability. This approach not only can accurately identify whether an application is over-privileged with certain confidence level, but also validates the need of requesting access to the permissions declared in application's manifest. Our approach is orthogonal to other solutions, and can be used in order to compute mobile applications attack surface and the risk introduced by over-privileges.

Acknowledgments

The authors would like to thank the anonymous reviewers and David Shaw for their valuable comments.

Appendix A. Supplementary data

Supplementary data related to this article can be found at http://dx.doi.org/10.1016/j.cose.2014.10.005.

REFERENCES

Ali K, Lhoták O. Application-only call graph construction. In: Proceedings of the 26th European Conference on Object-Oriented Programming. Springer-Verlag; 2012. p. 688–712.

A tool for reverse engineering android apk files [Online]. Available https://code.google.com/p/android-apktool/.

Androguard - reverse engineering, malware and goodware analysis of android applications and more (ninja !) – google project hosting. [Online]. Available http://code.google.com/p/androguard/.

Android and security: Google bouncer. [Online]. Available http://googlemobile.blogspot.it/2012/02/android-and-security.html.

ASM – home page." [Online]. Available http://asm.ow2.org/.

Au KWY, Zhou YF, Huang Z, Lie D. Pscout: analyzing the android permission specification. In: Proceedings of the 19th ACM Conference on Computer and Communications Security. ACM; 2012. p. 217–28.

Barr K, Bungale P, Deasy S, Gyuris V, Hung P, Newell C, et al. The vmware mobile virtualization platform: is that a hypervisor in your pocket? SIGOPS Oper Syst Rev Dec. 2010;44(4):124–35.

Barrera D, Kayacik HG, van Oorschot PC, Somayaji A. A methodology for empirical analysis of permission-based security models and its application to android. In: Proceedings of the 17th ACM Conference on Computer and Communications Security. ACM; 2010. p. 73–84.

Bartel A, Klein J, Le Traon Y, Monperrus M. Automatically securing permission-based software by reducing the attack surface: an application to android. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ACM; 2012. p. 274–7.

Bartel A, Klein J, Le Traon Y, Monperrus M. Dexpler: converting android dalvik bytecode to jimple for static analysis with soot. In: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis. ACM; 2012. p. 27–38.

Beresford AR, Rice A, Skehin N, Sohan R. Mockdroid: trading privacy for application functionality on smartphones. In: Proceedings of the 12th Workshop on Mobile Computing Systems and Applications. ACM; 2011. p. 49–54.

Berthome P, Fecherolle T, Guilloteau N, Lalande J-F. Repackaging android applications for auditing access to private data. In: Proceedings of the 7th International Conference on Availability, Reliability and Security. IEEE Computer Society; 2012. p. 388–96.

Bishop M. Computer security: art and science. 1st ed. Addison-Wesley Professional; dec 2002.

Dex2Jar – modify apk with dex-tools – tools to work with android.dex and java.class files – google project hosting. [Online]. Available http://code.google.com/p/dex2jar/wiki/ModifyApkWithDexTool.

Egele M, Kruegel C, Kirda E, Vigna G. PiOS: detecting privacy leaks in iOS applications. In: Proceedings of the Network and Distributed System Security Symposium (NDSS); February 2011.

Enck W, Gilbert P, Chun B-G, Cox LP, Jung J, McDaniel P, et al. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. USENIX Association; 2010. p. 1–6.

FBI warns loozfon, FinFisher mobile malware hitting android phones," Oct. 2012. [Online]. Available http://www.networkworld.com/community/blog/fbi-warns-loozfon-finfisher-mobile-malware-hitting-android-phones.

Felt AP, Chin E, Hanna S, Song D, Wagner D. Android permissions demystified. In: Proceedings of the 18th ACM Conference on

Computer and Communications Security. ACM; 2011. p. 627–38.
Felt AP, Ha E, Egelman S, Haney A, Chin E, Wagner D. Android permissions: user attention, comprehension, and behavior. In: Proceedings of the 8th Symposium on Usable Privacy and Security. ACM; 2012. 3:1–3:14.
Feth D, Pretschner A. Flexible data-driven security for android. In: Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability. IEEE Computer Society; 2012. p. 41–50.
Fuchs AP, Chaudhuri A, Foster JS. Scandroid: automated security certification of android applications. Tech Rep 2009. Available http://www.cs.umd.edu/~jfoster/papers/cs-tr-4991.pdf.
Gibler C, Crussell J, Erickson J, Chen H. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In: Proceedings of the 5th International Conference on Trust and Trustworthy Computing. Springer-Verlag; 2012. p. 291–307.
Gu G, Fogla P, Dagon D, Lee W, Skorić B. Measuring intrusion detection capability: an information-theoretic approach. In: Proceedings of the ACM Symposium on Information, Computer and Communications Security. ACM; 2006. p. 90–101.
I. Apple, “ios.” [Online]. Available http://www.apple.com/ios/.
I. Google, “Andoid operating system.” [Online]. Available http://source.android.com/.
Jad java decompiler. [Online]. Available http://varaneckas.com/jad/.
Kaspersky security bulletin 2012. the overall statistics for 2012.” [Online]. Available https://www.securelist.com/en/analysis/204792255/Kaspersky_Security_Bulletin_2012_The_overall_statistics_for_2012.
Kodeswaran P, Nandakumar V, Kapoor S, Kamaraju P, Joshi A, Mukherjea S. Securing enterprise data on smartphones using run time information flow control. In: Proceedings of the 13th International Conference on Mobile Data Management. IEEE Computer Society; 2012. p. 300–5.
Miller, C. and Oberheide, J. Dissecting the android bouncer. [Online]. Available http://jon.oberheide.org/blog/2012/06/21/dissecting-the-android-bouncer/.
Mobithinking, “Global mobile statistics 2013 part a: Mobile subscribers; handset market share; mobile operators.” [Online]. Available http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats/a#subscribers.
Mobile apps take data without permission. [Online]. Available http://bits.blogs.nytimes.com/2012/02/15/google-and-mobile-apps-take-data-books-without-permission/.
Orthacker C, Teufl P, Kraxberger S, Lackner G, Gissing M, Marsalek A, et al. Android security permissions â€“ can we trust them?. In: Proceedings of Security and Privacy in Mobile Information and Communication Systems, vol. 94. Springer Berlin Heidelberg; 2012. p. 40–51.
Pandita R, Xiao X, Yang W, Enck W, Xie T. Whyper: towards automating risk assessment of mobile applications. In: Proceedings of the 22nd USENIX Conference on Security. USENIX Association; 2013. p. 527–42.
P. Ducklin, “Apple's app store bypassed by russian hacker, leaving developers out of pocket.” [Online]. Available http://nakedsecurity.sophos.com/2012/07/14/apple-app-store-bypassed-by-russian-hacker-leaving-developers-out-of-pocket/.
Portokalidis G, Slowinska A, Bos H. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. In: Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006. ACM; 2006. p. 15–27.
Rastogi V, Chen Y, Enck W. Appsplayground: automatic security analysis of smartphone applications. In: Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy. ACM; 2013. p. 209–20.
Rosen S, Qian Z, Mao ZM. AppProfiler: a flexible method of exposing privacy-related behavior in android applications to end users. In: Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy. ACM; 2013. p. 221–32.
Schreckling D, Kstler J, Schaff M. Information Security Technical Report. Kynoid: real-time enforcement of fine-grained, user-defined, and data-centric security policies for android, vol. 17; Feb 2013. p. 71–80. no. 3.
Shabtai A, Fledel Y, Kanonov U, Elovici Y, Dolev S, Glezer C. Google android: a comprehensive security assessment. Secur Priv IEEE 2010;8(2):35–44.
Stephan H, Siegfried R. Javascript in android apps â€“ an attack vector. 2013 [Online]. Available, http://www.bodden.de/2013/09/16/javascript-in-android-apps-an-attack-vector/.
Stevens R, Ganz J, Filkov V, Devanbu P, Chen H. Asking for (and about) permissions used by android apps. In: Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press; 2013. p. 31–40.
Stirparo P, Kounelis I. The mobileak project: forensics methodology for mobile application privacy assessment. In: Proceeding of the International Conference on Internet Technology and Secured Transactions. IEEE Press; 2012. p. 297–303.
Vallee-Rai R, Hendren LJ. Jimple: Simplifying java bytecode for analyses and transformations. 1998 [Online]. Available, http://www.sable.mcgill.ca/publications/techreports/sable-tr-1998-4.ps.
Vallée-Rai R, Co P, Gagnon E, Hendren L, Lam P, Sundaresan V. Soot - a java bytecode optimization framework. In: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research. IBM Press; 1999. p. 13.
Wei X, Gomez L, Neamtiu I, Faloutsos M. Permission evolution in the android ecosystem. In: Proceedings of the 28th Annual Computer Security Applications Conference. ACM; 2012. p. 31–40.
Werthmann T, Hund R, Davi L, Sadeghi A-R, Holz T. Psios: bring your own privacy and security to ios devices. In: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security. ACM; 2013. p. 13–24.
Xiao X, Tillmann N, Fahndrich M, De Halleux J, Moskal M. User-aware privacy control via extended static-information-flow analysis. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ACM; 2012. p. 80–9.
Xing L, Pan X, Wang R, Yuan K, Wang X. Upgrading your android, elevating my malware: privilege escalation through mobile os updating. In: Proceedings of the 35th IEEE Symposium on Security and Privacy. IEEE Computer Society; 2014. p. 393–408.
Yan LK, Yin H. Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In: Proceedings of the 21st USENIX Conference on Security Symposium. USENIX Association; 2012. p. 29.
Permission explorer. [Online]. Available https://play.google.com/store/apps/details?id=com.carlocriniti.android.permission_explorer&hl=en.

Dimitris Geneiatakis holds a Ph.D. in the field of Information and Communication Systems Security from the Department of Information and Communications Systems Engineering of the University of Aegean, Greece. His current research interests are in the areas of security mechanisms in Internet telephony, smart cards, intrusion detection systems, and network and software security. Currently, he is postdoctoral researcher at Joint Research Centre of European Commission. Previously, was within Columbia University as a postdoctoral researcher. He is an author of more than thirty refereed papers in international journals and conference proceedings.

Igor Nai Fovino holds a Ph.D. in computer security. His research fields belong to the area of ICT Security of industrial systems and Smart Grids, Intrusion Detection Techniques, Cryptography and Secure Network Protocols. He is an author of more than sixty scientific papers published on international journals, books and conference proceedings. He is member of the IFIP Working group 11.10 for the Protection of Critical Infrastructures and serves as International Expert within the ERNCIP European Expert Group on the security of Energy Smart Grids. Currently, he is within the Joint Research Centre of the European Commission as a scientic project manager.

Ioannis Kounelis is an ICT security researcher working at the Joint Research Centre of the European Commission. He is in parallel a Ph.D. student at the Royal Institute of Technology - KTH in Stockholm, Sweden. He has received his Master of Science in Computer Security in 2010 from KTH, while in 2007 he received his Bachelor of Science in Computer Science from the Aristotle University of Thessaloniki, Greece. His main research activities focus on mobile security and secure system design.

Pasquale Stirparo is a Ph.D. student at the KTH - Royal Institute of Tech-nology in Stockholm and holds a MSc in Computer Engineering from Politecnico di Torino. His research interests revolve around digital forensics and the security and privacy issues related to mobile devices communication protocols. Currently, he is working as Digital Forensics and Mobile Security Researcher at the Joint Research Centre of the European Commission. Prior to join JRC, Pasquale was working as Security Consultant and Digital Forensics Analyst for an Italian based private company. He has also been invited as speaker to several Italian conferences and seminars on Digital Forensics.