European Commission

JRC TECHNICAL REPORT

Mobile Applications Privacy

Towards a methodology to identify over-privileged applications

Igor NAI FOVINO
Ricardo NEISSE
Dimitris GENEIATAKIS
Ioannis KOUNELIS

2014

Report EUR 26484 EN

Joint Research Centre

European Commission
Joint Research Centre
Institute for the Protection and Security of the Citizen

Contact information
Igor Nai Fovino
Address: Joint Research Centre, Via Enrico Fermi 2749, TP 361, 21027 Ispra (VA), Italy
E-mail: igor.nai-fovino@jrc.ec.europa.eu
Tel.: +39 0332785809

JRC Science Hub
https://ec.europa.eu/jrc

JRC87818

EUR 26484 EN

ISBN 978-92-79-35409-0

ISSN 1831-9424

doi:10.2788/66345

Luxembourg: Publications Office of the European Union, 2014

Abstract

Smart-phones are today used to perform a huge amount of online activities. They are used as interfaces to access the cloud, as storage resource, as social network tools, agenda, digital wallet, digital identity repository etc. In other words smart-phone are today the citizen's digital companion, and, as such, they are the explicit or implicit repository of a huge amount of personal information. The criticality of these devices is generally due to the following considerations: 1. Being mobile by nature, they are exposed full-time to a potentially adverse environment; 2. The need,for mobile applications, to cut the development costs to maintain the price appealing for the mobile-application market, is often translated into a quick prototyping approach, rather than a careful cyber-security oriented code development; 3. Being the smart-phone strongly linked to their owner, a successful exploitation of a smart-phone can directly impact the security and privacy of its owner. One of the major source of back-doors of mobile applications, is the bad use of privilege permissions. Developers tend to attribute to their applications as much permission rights as possible, even if they are not indeed needed.Malicious applications can leverage of these permissions to create covert channels allowing to get private information stored into the smart-phone. In this report we investigate on the "Declarative permissions scheme model" on which relies the security layer of Android, proposing an innovative technique combining together dynamic and static analysis to profile mobile applications and identify if they are over-privileged. In the same report we introduce also a first proposal for

1 Executive Summary

Smart-phones are today used to perform a huge amount of online activities. They are used as interfaces to access the cloud, as storage resource, as social network tools, agenda, digital wallet, digital identity repository etc. In other words smart-phone are today the citizen's digital companion, and, as such, they are the explicit or implicit repository of a huge amount of personal information. The criticality of these devices is generally due to the following considerations:

1. Being mobile by nature, they are exposed full-time to a potentially adverse environment
2. The need, for mobile applications, to cut the development costs to maintain the price appealing for the mobile-application market, is often translated into a quick-prototyping approach, rather than a careful cyber-security oriented code development
3. Being the smart-phone strongly linked to their owner, a successful exploitation of a smart-phone can directly impact the security and privacy of its owner

One of the major source of back-doors of mobile applications, is the bad use of privilege permissions. Developers tend to attribute to their applications as much permission rights as possible, even if they are not indeed needed. Malicious applications can leverage of these permissions to create covert channels allowing to get private information stored into the smart-phone.

In this report we investigate on the "Declarative permissions scheme model" on which relies the security layer of Android, proposing an innovative technique combining together dynamic and static analysis to profile mobile applications and identify if they are over-privileged. In the same report we introduce also a first proposal for enforcing the end-user control on the hidden behaviours of mobile applications.

This report is the first of a series in which privacy and security aspects of smart-phones will be analysed.

2 Introduction

In this report we investigate mobile applications (apps in the following) in order to identify potential weaknesses that lies on the exploitation of needless privileges. Mobile applications are the instruments through which the user interacts with his mobile device. They have, often, access to all sorts of personal data (*e.g.,* pictures, private messages, bank accounts, credit cards, *etc.*). For that reason their security should be highly taken into account and enforced as much as possible, to protect the end-user from privacy breaches and security threats. Although mobile applications in general leverage on some of the native security functionalities of the underlying operating system in order to guarantee a minimum level of security, the business model on which the development process of mobile applications is based (aiming at minimizing development costs and sw life cycle), implicitly influence the amount of resources invested by developers in taking care of the security of their apps. As a result, it is not rare to see on the market applications that can be easily mis-configured by malicious actors to obtain unauthorized personal information. This can be for example the case when mobile applications ask for permissions which they do not need for their execution, permissions that, in the following, might be exploited by malicious apps to perform unauthorised actions. The over-permission of mobile apps, as will be showed in this report, is a quite common practice among apps developers, but it is also a possible source of threats that can have a direct impact on the users privacy. It is of high importance to investigate mobile application for potential privacy and security flaws especially taking into consideration the almost pervasive presence of mobile applications in our daily life. Scope of this report is that of presenting the first results of a new technique we developed to profile mobile applications, with the aim of identifying if a given mobile app is over-priviledged and consequently prone to possible miss-configurations and attacks.

The mobile applications business model is based on a one-stop shop model on which the app-stores (*Google play store, Apple store (iOS)*, *etc.*), allow to 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 [1]. Though users trust these centralized stores and their security approaches, it is almost impossible to be 100% secure of the correctness of any given application. For instance, in [2] is presented a technique to bypass *Google's* Bouncer security checks. A similar problem was faced also by *Apple's store* [3].

These threats acquire a high relevance on the light of the fact that 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 the smartphones an interesting target for attackers. For example, spying applications can collect user's position or steal personal information and sell them to marketing companies [4]. Even well-known applications may manipulate their access to personal information as shown in various research works [5–7]. In other cases a mobile application might be *over-privileged*; meaning that it requests more permission 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 [8] and provide access to otherwise private information.

Further, end-users might try to install applications from third party stores, which do not scrutinize the functionality of the provided applications. These facts show that even the existence 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 that is handled by the provided applications.

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 application are analyzed to identify possible sources and sinks of data leakages (*e.g.,* [9–11]) without executing it.
- Dynamic monitoring: The behavior of applications is examined at runtime (*e.g.,* [6, 12]).
- Scanning applications: Third party applications, like *Permission Explorer* [13], are able to scan all the installed application and generate a user friendly report notifying users for the usage of the requested permissions.
- Operating systems privileges enforcements: Operating systems enforce specific mechanisms in order to eliminate personal data manipulation. For instance, Android OS requests from the user to give explicit authorization access to 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.,* [9–11]), usually do not take into account the runtime context, making them prone to false negative identification or requiring, to be effective, the source code access or/and modification to the underlying framework. For instance, [10] modifies the Android framework to log the permission checks, while solutions such as [6] do not focus on identifying over-privileged applications.

On the light of these considerations, in this report, we elaborate on identifying *over-privileges*, and validating the need of declaring specific permissions in the manifest of any given Android application, by combining static analysis and runtime information. With such an approach, we aim at combining the advantages of exhaustive static analysis with the discrimination power of dynamic analysis, to provide to the end-user, an useful instrument to limit and restrict the freedom degrees of mobile applications of uncertain provenance.

We focus on the Android OS because it is among the most utilized operating systems in the market, and it is considered a main target for attackers [14]. Particularly, we capitalize on the advantages of Dexpler [15] and Soot [16] framework to reverse engineer and analyze any given Android application both statically and dynamically. We record and instrument all the possible Application Programming Interfaces (APIs) identified at the reversed engineered code in order to monitor the APIs executed at runtime as well. Relying on the extracting information we audit the permission included in the application's manifest.

To the best of our knowledge, this is the first work focusing on determining *over-privileges* on Android applications by linking static and runtime information. This approach complements other solutions such as [9–12], and can accurately *justify* whether

or not the application request to a specific resource is required. Results show that 66% (8/12) of the examined applications were identified as *over-privileged*. The main contributions of this work consists in the definition of a new approach allowing to identify *over-privileged* applications in Android OS and enabling the explicit validation of the un-required permissions by the user. This approach has, moreover, the advantage of being transparent with respect to the OS and the application's source code (that is, in fact, not needed to perform the analysis). It is worth to note that the developed prototype is freely available[1].

The rest of this report is structured as follows. In Section 3 we provide an overview of the Android OS security model and we describe the security issues that introduced by over-privileged applications. In Section 5 we outline our approach for identifying *over-privileged* applications by combining static and dynamic analysis information, while in Section 6 we evaluate this approach. We overview other similar works in Section 7 and we comment on the findings of our approach in Section 8. In Section 9 we illustrate the limitations of our approach and present some pointers to future work. Finally, in Section 11 we draw our conclusions.

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

3 Preliminaries

In this section an overview of the cyber-security aspects of Android and mobile applications is provided.

3.1 An Overview of Android Security Model

The core of the Android 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 applications. Figure 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 1.1[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.

```
<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 1.1: An example of a real Android application manifest records. The application requests access to various resources such as *Camera, Internet, Calendar, etc.*

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

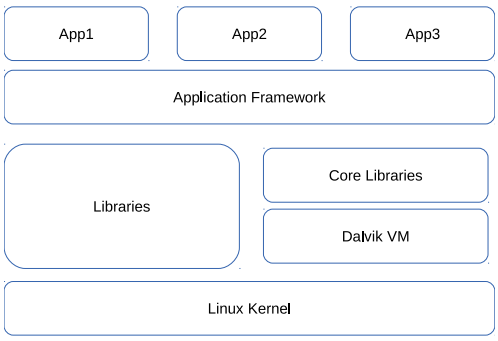


Fig. 1: Android software modular architecture.

3.2 Permission Back-doors

Mobile users enjoy the advantages of mobile operating systems evolution combined with high speed network access, which mobile operators offers, albeit the fact that new threats are emerged. As mobile applications manage a wide range of personal information such as unique identifiers, location, call history, text messages, emails, *etc.*, they generate new opportunities for profiling users' and manipulating these data in return of financial benefit. Not only malicious applications (malware) misuse such information, but even legitimate ones. This is, for instance, the case of the *Twitter application* which sent out users' personal information, without notifying the users [17] beforehand. This information can be even lost or modified if applications are allowed to execute the corresponding operations.

In other cases, malware might exploit legitimate applications' configuration vulnerabilities, and/or manipulate their permissions in order to gain access either to private information or to other protected functionalities, provided only to the legitimate applications. This, for instance, can be achieved through inter-process communication as demonstrated in [8], without the need to exploit a vulnerability. Alternatively, a malware might exploit a specific vulnerability that will allow the execution of an API that otherwise was not able to be executed, as illustrated in Figure 2. *Webview* , for example, is vulnerable to malicious input, as referred in [18]. In that case the malware can execute any API, if the exploited application has the appropriate permissions. A detailed analysis of personal data manipulation in mobile applications can be found in [5–7].

It should be noted that these problems are mainly due to the fact that the Android permission's system assumes that an application can only use the functionality for which the appropriate permissions are available.

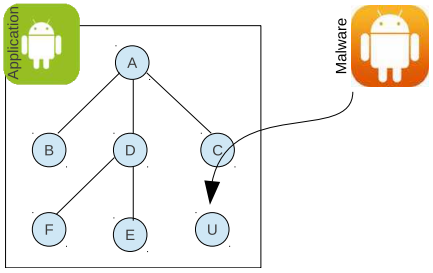


Fig. 2: Example of exploiting vulnerabilities, by a malware, on the method *C* in order to execute the method *U* that is "out of the scope" of the application. By the term "out of the scope" we mean that the malware is taking the advantage of the application's needless privileges to execute the method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

4 Profiling Techniques for Android Applications

The techniques that can be deployed to monitor and determine applications' behaviour can rely either on static or dynamic analysis depending on:

1. the application's source code availability, and
2. the applications properties and features needed to be studied.

In this Section, we overview the dynamic analysis tools that can be used for profiling different features of Android applications. Note that currently we focus only on dynamic analysis tools, since we are interested in the information that can be extracted from an Android application, during runtime in order to assess its security level.

Dalvik Debug Monitor Server: In case in which applications' source code is available, the Dalvik Debug Monitor Server (DDMS) [19] can be used in order to monitor various metrics of the methods that we are interested in. For instance, relying on DDMS we can extract information such as the called methods, the number of calls to other methods, the time spent by the examined method, and other related information. To extract this data the application developer should include the `startMethodTracing()` and `stopMethodTracing()`, to start and terminate the monitoring, in the application's source code. APIs

Operating System Level Monitoring: The Systrace [20] can be used to monitor other properties of the examined application at the operating system level such as mobiles cpu usage, memory and other information. Note that systrace also can be used to trace specific parts of the application code by including the `Trace.beginSection()` and `Trace.stopSection()` in the parts of the code that the developer would like to monitor.

TaintDroid is a solution which applies on Dalvik bytecode. It requires the modification of Android OS in order to monitor if applications manipulate private information [6]. TaintDroid achieves this goal by marking data originated from predefined sensitive sources, such as global positioning system, and monitoring their flow during execution at the operating system level.

Repackaging and Instrumentation: Android mobile applications target the Dalvik, instead of pure Java, bytecode. Java based applications can be reverse-engineered using tools such as Jad [21] and ASM [22]. Similarly, the android mobile applications can be analyzed and modified using tools such as ApkTool [23], Androguard [24], Dexpler [15] and Dex2Jar [25]. Note that in most cases Dalvik bytecode is not translated to the original Java code but to an intermediate format, depending on the tools capabilities. As soon as the analysis is completed the application is signed and can be installed and executed on the Android phone. The general procedure for analyzing and repackaging an Android application is illustrated in Figure 3.

In the following we provide an overview of the mentioned tools.

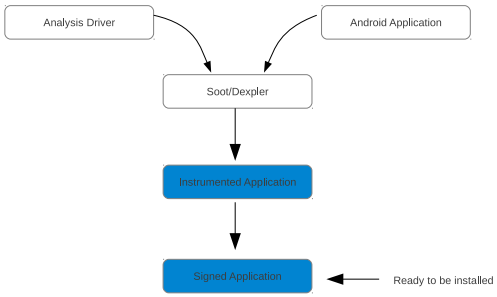


Fig. 3: A high level procedure for repackaging android applications.

DroidBox: The Droidbox [26] is a dynamic analysis tool that enables automatically to monitor information such as net read, net write, cryptographic functions and other operations. The current version of DroidBox does not support the monitor of all the available methods provided by the Android OS. It can monitor only predefined methods declared in a configuration file.

Dexpler: Dexpler [15] enables the transformation of Dalivk bytecode to Jimple representation; a Soot's [16] intermediate format. This way, existing Soot analysis tools and libraries can be re-used, while at the same time additional tools can be built on top of the Soot to analyze, modify and repackage Android applications.

ApkTool: Similar to the Dexpler, the ApkTool [23] analyzes, modifies and repackages Android mobile applications. The ApkTool does not support the Jimple intermediate representation, as Dexpler does, but converts the Dalvik bytecode to another intermediate format named Smali [27].

Dex2Jar: The Dex2Jar [25] is a repackage tool, such as Dexpler and ApkTool. The Dex2Jar converts the Dalvik bytecode to Jasmin [28], an assembly format for Java, in which you can introduce your code and create a new application afterwards. This functionality is provided by the Dex2Jar through the Dex-reader,translator,ir and tools components.

Androguard: The Androguard [24] is another tool for reverse engineering Android applications statically. The current version of Androguard provides different features such as checks if an application belongs to a malware database, integration with external decompilers, and access to static analysis information (basic blocks, instructions, *etc.*).

5 Proposed Approach

As described in the introduction, we are interested in defining a method allowing to effectively profile and analyse mobile applications, in search for over-priviledges. 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] by a target application. This approach requires neither access to the applications' original source code nor modification of the underlying framework.

We rely on static analysis to compute the (maximum) set of permissions that might be used by the examined application, while we validate their proper employment relying on dynamic analysis.

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.

5.1 Application Analysis

In this work, we rely on Dexpler [15] a Soot framework [16] based tool for analyzing, modifying and repackaging android mobile applications as mentioned previously. Our choice is based on the fact that Soot framework provides ready to use libraries for analyzing both statically and dynamically an Android application. In this framework, any given android application can be combined with a proper designed analysis driver to analyze and introduce new code to the initial application; the outcome is the instrumented application.

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 [29] through dexpler, which enables the usage of Soot framework [16]. 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 1.2.

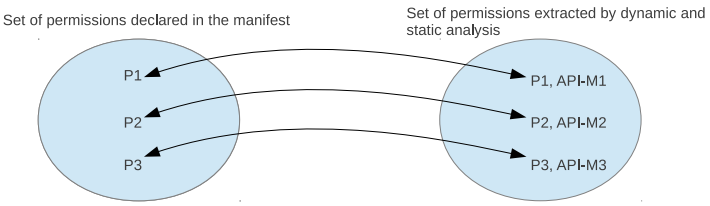
As soon as the analysis is completed the application is signed and can be installed and executed on the Android phone. The general procedure for analyzing and repackaging an Android application is illustrated in Figure 3.

5.2 Identify over-privileged applications

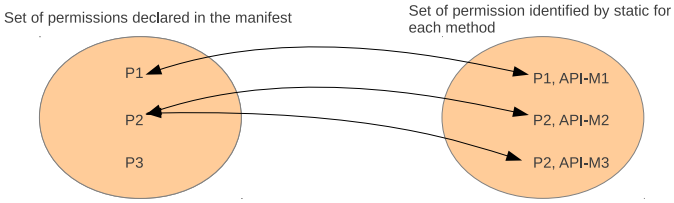
An application is considered *over-privileged* if and only if there is a permission record in the manifest without matching to any of the permissions identified in the static analysis part. Complementary, an application is validated as non over-privileged, if and only if the static, dynamic and manifest permission sets match. These cases are illustrated in Figure 4, while the whole procedure to identify an *over-privileged* application is illustrated in Figure 5. To determine *over-privileged* applications, we integrated the Dexpler [15] and Soot framework [16] with an analysis driver that:

1. Identifies and records all the methods existing in the reversed engineered application.

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



(a) A validated non over-privileged set.



(b) An over-privileged set.

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

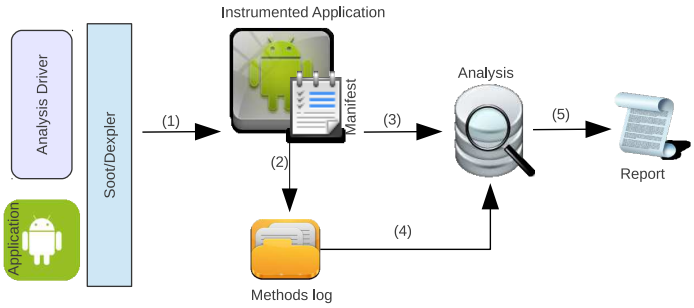


Fig. 5: Proposed run-time verification approach.The soot framework analyzes the mobile application and inserts small pieces of code to monitor the executed methods (1). Every time the application is executed all the methods are logged (2) and analyzed (3,4) in order to determine whether the application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

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. This allows us to run the application without the need to modify the manifest of the original application.

The output of the analysis generates a new android application package, the instrumented one, which records (a) all the possible methods that exist in the reversed engineered application, and (b) the called methods at runtime. We call the former as the static analysis part of our approach. Note that the reversed engineered code does not necessarily correspond to the application's original source code. The instrumented application should be executed manually to reproduce common real life situations and record the called methods. We are considering to create an automated procedure in order to navigate through all the functionalities of the targeted Android application for future work. 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 (the outcome of the static analysis), and
2. The permission set required for the examined applications execution (the outcome of the dynamic analysis).

This is achieved by identifying the signature of the executed call in the permission mapping database, based on the permission mapping published in [10]. Afterwards, 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.*

We control whether the set of static analysis is a subset of the permissions declared in the manifest or not. If so then we deduce that the application is *over-privileged* with accuracy of 100%. This is because the static analysis set is the superset of the permissions that the application need to be executed, and consequently the manifest set cannot be a superset of the static analysis. If not we examine the set of permissions extracted at the dynamic analysis (runtime information) and compare it with the manifest's and the static analysis permission sets. We deduce that the application is not *over-privileged* with accuracy of 100% if these sets are equal, as the runtime information matches with static analysis outcome. This means that all the methods, requesting specific permission to be executed, were reached during runtime.

In the case that the conjunction of runtime analysis and manifest permission sets is a superset of the conjunction of the static analysis and manifest permission sets there is a possibility of manipulating the examined application's permissions. However, this can also be a dynamic analysis false identification[4]. Note that if the static analysis set is a superset of the manifest we analyse only the manifest set, since all the other permissions will not be triggered through an API as the Android OS will throw a security exception. Combining in that way static and dynamic analysis we guarantee:

1. The need of declaring specific permission in the manifest.
2. The identification of useless permissions declared in the manifest.
3. Whether a given application is*over-privileged* or not.

[4] In the context of this work, we refer to false positive as the case where permission identified by our approach as used are not, in reality, used at all. The opposite is the case for false negatives.

```
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
}
```

Listing 1.2: An Example of analysing a mobile application with Soot framework for identifying a particular method such as getDeviceId.

6 Evaluation

To demonstrate and evaluate our approach's effectiveness we performed a first initial campaign where we analyzed twelve Android applications belonging in different categories in order to classify them as over-privileged or not. We are planing to analyze additional applications in a future work. For the purpose of this initial analysis, we distinguish the examined application in the following categories based on their functionalities:

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.

To extract the runtime information, which includes the called methods, we executed manually each of the analyzed applications twenty times under different test case scenarios, while the static analysis information was generated during instrumentation. Outcomes show that most of the examined applications are *over-privileged*; meaning that they request and gain access to permissions which are not needed for their execution. Table 1 summarizes the outcomes of our analysis by identifying whether or not the examined application is *over-privileged*.

Table 1: Types of applications analysed for over-privileges identification. We collect static analysis and runtime information. In particular, we identify both the number and the signature of the methods that exist in the reversed engineered applications, and we provide the statistics related to the executed methods. We determine that eight out of the twelve examined applications as over-privileged.

Type	Methods	Executed Methods Max/Avg.	Over-privileged
Expenses(1)	1830	661/558	Yes
Entertainment(1)	1873	32/31	No
Accessories(1)	559	147/113	Yes
Expenses(2)	3910	1172/925	Yes
Linguistic	2105	667/605	No
Shopping(1)	1848	631/596	Yes
Hello	2113	18/18	No
Shopping(2)	505	183/136	No
Shopping(3)	257	169/125	Yes
Shopping(4)	211	141/132	No
Entertainment(2)	2180	98/74	Yes
Accessories(2)	396	209/138	No

Our approach determined that eight out of twelve (66%) of the examined applications request more permissions than those needed for their execution, whereas only 4 out of 12 (34%) of the examined applications are validated as not *over-privileged*. The most common permissions the *over-privileged* applications request are illustrated in Table 2, while Table 3 shows the permissions requested and not used by the examined applications, combining the results of static and runtime analysis.

As 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 [30]. This is also the case for the Android applications relying on Java as indicated in our results (refer to Table 1).

Table 2: Most used permissions among over-privileged applications as identified by our approach.

Permission Type	Usage by Application
WRITE_EXTERNAL_STORAGE	37.5%
RECEIVE_SMS	25%
READ_CONTACTS	50%
SEND_SMS	50%
READ_PHONE_STATE	50%

It should be noted that by exploiting the advantages of runtime information, on one side, our approach does not generate false positive identification alarms that static analysis might generate. This is due to the fact that an application in order to execute a *method* that requires specific permissions has to get authorization during installation, otherwise the execution will fail whenever this method is triggered. On the other side, the static analysis eliminates the false negatives which the dynamic analysis might trigger. Though the number of the examined applications are limited, results show that our approach can be used not only to prove that a declared permission is indeed been used, but also to identify *over-privileged* applications.

7 Related Work

In this section we mainly overview the works focus on the elimination of users' privacy violations for the Android and iOS operating systems as they are the most used in the market. To eliminate the risk of personal data manipulation Android and iOS operating systems follow different approaches. On the one hand, Android OS [31] 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 capacity to users for dynamically enabling permissions. On the other hand, iOS [32] 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.,* [2, 32]), but, even worst, "benevolent" applications can manipulate personal data as demonstrated in [6,7]. In this context, [33] introduces a methodology based on self-organized maps for assessing Anrdoid's permission model, while the PScout solution [34] develops a tool for assessing permissions of the Android OS by statically analyzing its source code. The security level of the Android OS is criticized in [35], while the AppsPlayground [36] introduces a framework for automated dynamic security analysis of Android applications.

Various researches are working 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. [6] 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, while [37] allows users to revoke access to particular resources at run-time. Similarly, solutions such as [38, 39] deploy a run time monitor for enabling users to control their data through their defined policies. The work presented in [40] introduces on [6] the notion of fine grained security policies to monitor applications' behaviour. Analogous research works have been accomplished for iOS [41,42]. To avoid the modifications in the underlying framework (*e.g.,* middleware, OS,*etc.*) [12] proposes repackaging of the application in which the compiled applications are analyzed and injected with particular code at the bytecode level in order to monitor all the access of personal data.

In [7, 43] are introduced alternative approaches based on static analysis to classify the information flows inside the application as safe or unsafe in terms of privacy. ScanDroid [44] 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 been tested in real-world applications yet. In [11] the authors develop a knowledge base of privacy related behaviours, which is used to assess the privacy "level" of a given application.

Besides the techniques used to eliminate the privacy violations by controlling users data, other works focus on detecting *over-privileged* applications [9, 10]. These works rely on application static analysis in order to identify over-privileges for any given ap-

plication. Each of these solutions develops a permission map first and then use a static analysis approach to identify possible data leaks, however, the permission map is published only in [10]. In [45] the authors accomplished a thorough survey to examine the effectiveness of the permission system in terms of supporting users to take the appropriate security decisions based on their needs.

8 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 applications by combining static and runtime analysis. Note that the Android OS uses the permissions as a mechanism to protect access to ”sensitive“ APIs. This way, if an application follows the least privilege principle [46] a potential exploitation would have a minimum impact. However, the existence of needless permissions offer the chance to bypass this protection mechanism.

The outcomes reveal that applications request access to permissions that are not required for carrying out their tasks. Consequently, this can be of high risk, because malware might discover such parts of the code in applications and manipulate it for accessing personal data, or some other functionality, without users being able to recognize such malicious activities. Table 2 illustrates the needless permissions that applications request. For instance, the examined *over-privileged* applications request access to permissions such as the SEND_SMS and the WRITE_EXTERNAL_STORAGE. Malware might exploit these permissions not only to gain access to otherwise private information, but also to profit by sending SMS to premium rate services.

Exploiting the advantages of runtime information, on the one side, our approach does not generate false positive identification alarms that static analysis might generate. This is due to the fact that an application, to execute a *method* that requires specific permissions, has to get authorization during installation, otherwise the execution will fail whenever this method is triggered. On the other side, the static analysis eliminates the false negatives, which the dynamic analysis might trigger. Nevertheless, it should be noted that dynamic analysis is an alternative and complementary approach technique to static analysis for determining *over-privileged* applications. As the results show, in most cases the dynamic analysis' outcome set is a superset of the static analysis. This is because, we execute the applications manually, and thus we cannot guarantee the coverage of all the possible paths. We validate with accuracy the need of the declared permissions in the manifest by combining the static and runtime information.

In addition, relying on the proposed solution, the false positive alarms generated by other static analysis approaches can be eliminated. For instance, in Stowaway [10], a solution that relies on static analysis, authors mention that their approach generates false positives. This is due to the fact that Stowaway does not take into account which parts of the code are executed. We determined such a case when we use Stowaway for analyzing the *Hello* application. In details, Stowaway identifies the permission WRITE_EXTERNAL_STORAGE unnecessary, and characterizes this application as *over-privileged*. However, the application needs this specific permission to execute a write operation in the external storage. We are aware of this since we develop 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 in the *Hello* application we use this particular API, and the Stowaway analysis online tool considers the WRITE_EXTERNAL_STORAGE permis-

sion as an extra permission. We should note that we compared the outcomes of this work only with the Stowaway solution because no other solution provides the code or an online service for analyzing application's permissions. Table 3 overviews the comparison between the Stowaway and our approach.

Further, 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 carrying out any type of analysis analysis on it. Consequently, they do not provide any valuable information on how the examined applications' declared permissions are used.

We should note that if an application is not over-privileged does not necessary mean that it is not malware. An application can be infected by 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) bad programming techniques from the developers side, and (b) potential points of manipulation. Our goal in this phase is to focus our research only on the over-privileges and use the findings for future activities.

Table 3: An *over-privileged* accuracy comparison between Stowaway [10] and our dynamic and static analysis approach.

Type	Stowaway [10]	Dynamic analysis permission set	Static analysis permission set
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
Expenses(2)	WRITE_EXTERNAL_STORAGE	CAMERA READ_CALENDAR READ_CONTACTS RECEIVE_BOOT_COMPLETED VIBRATE WRITE_CALENDAR WRITE_EXTERNAL_STORAGE	CAMERA READ_CALENDAR RECEIVE_BOOT_COMPLETED WRITE_CALENDAR
Linguistic	—	—	—
Hello-World	WRITE_EXTERNAL_STORAGE	—	—
Shopping(2)	—	—	—
Shopping(3)	READ_SMS	READ_CONTACTS READ_SMS	READ_CONTACTS READ_SMS
Shopping(4)	—	—	—
Entertainment(2)	INTERNET READ_PHONE_STATE	ACCESS_NETWORK_STATE INTERNET READ_PHONE_STATE	INTERNET READ_PHONE_STATE
Accessories(2)	—	—	—

9 Limitations and Future Improvements

Our approach main limitation lies in the fact that the dynamic analysis part might generate false negative identifications, since we cannot guarantee a complete code coverage. This is due to the fact that, in the current evaluation, we rely on user's interaction and as a result some parts of the code might not be executed since particular criteria are not met. Thus, we are considering to build an extension of our solution that will achieve as much as possible coverage of the code surface. However, we eliminate this problem by exploiting static analysis information.

Further, 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 [15] as it does not handle optimized Dalvik (odex) opcodes. Even more 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 these cases in order to eliminate those problems.

10 Towards a Framework for Eliminating Privacy Violations in Mobile Applications

In order to enforce runtime security policies in an Android application we propose the framework described in Figure 6. In this framework an Android application is used as input with the list of Android permissions for the authoring of security policies (Policy Authoring). Policies specify what an application is allowed to do, what should be denied, and possible modifications on the application activities. For example, a security policy could be specified to obfuscate the user location of the phone unique identifier. The specified security policies are used as input for the instrumentation of the application (Security Instrumentation) to generate a new instrumented application that at runtime will support the enforcement of the policies. The instrumentation also inserts information flow tracking code, that initializes data identifiers and invokes an Information Flow Monitor application when data is received by the application from the environment of send to the outside environment.

When an instrumented application is running, all the executed actions that are referenced in the policy are redirected to the Security Enforcement Monitor application, which checks the security policies and possible executes enforcement actions. In Figure , one example instrumentation shows the `getDeviceIdEvent` being notified to the enforcement monitor app with the answer to modify the id to the value 0.

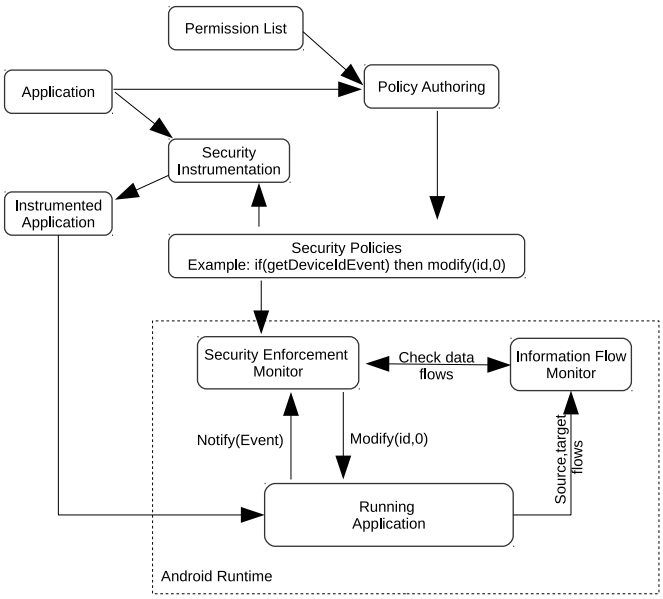


Fig. 6: A proposed framework for security policy enforce in Android Applications

Is part of our future work to identify and design efficient instrumentation and runtime monitoring techniques for Android applications. Furthermore, the specification of security policies should not rely on mobile phone users, and security policy templates to support secure mobile phone usage will be specified. In the current Android model users are unaware of the risks and are unable to verify the requested permissions of all installed applications.

11 Conclusions

Over-privileged applications constitutes a serious source of threat as they can be used as back-doors to get the access to personal information managed in smart-phones. Users have to trust the manifest declaration for any given application, on Android OS, if they would like to use it. However, even benevolent *over-privileged* applications might be exploited by malicious applications to gain access to otherwise un-accessible (personal) data. In this report, after having provided an overview of the problem and of the scientific literature on the topic, we introduced a new approach that combines static and dynamic analysis to identify *over-privileged* applications. This approach not only identifies accurately whether an application is *over-privileged*, but also provide means to analyse the behaviour of unknown mobile applications. The application of this analysis method during a preliminary test campaign, allowed to draw the following considerations:

- the technique developed is effective in identifying over-priviledged applications
- it can be used to perform a behavioural analysis of target applications
- the bad use, by the developers, of the privileges attributes schemes in mobile applications is quite common

The last consideration is, under a cyber-security perspective, extremely relevant. For this reason, we plan to conduct an extensive analysis campaign of general purpose mobile applications, to assess the real magnitude of the phenomenon.

What is also important to note is that, at the moment, the end-user is almost blind with respect to the hidden activities of the mobile applications installed on his smartphone. To truly secure the end-user sensitive personal information and to protect him from cyber-threats we believe that two actions would be needed:

1. The definition of an open framework allowing to give back to the end-user the full control on the access of every information and feature stored on his mobile phone
2. The definition of a set of best practices guiding the end-user in understanding how to better protect his online activities

A following set of reports will be delivered to cover these two main topics

References

1. "Android and security: Google bouncer." [Online]. Available: http://googlemobile.blogspot.it/2012/02/android-and-security.html
2. C. Miller and J. Oberheide, "Dissecting the android bouncer." [Online]. Available: http://jon.oberheide.org/blog/2012/06/21/dissecting-the-android-bouncer/
3. 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/
4. "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
5. P. Stirparo and I. Kounelis, "The mobileak project: Forensics methodology for mobile application privacy assessment," in *Internet Technology And Secured Transactions, 2012 International Conferece For*, 2012, pp. 297–303.
6. W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–6.
7. C. Gibler, J. Crussell, J. Erickson, and H. Chen, "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*, ser. TRUST'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 291–307.
8. C. Orthacker, P. Teufl, S. Kraxberger, G. Lackner, M. Gissing, A. Marsalek, J. Leibetseder, and O. Prevenhueber, "Android security permissions can we trust them?" in *Security and Privacy in Mobile Information and Communication Systems*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, R. Prasad, K. Farkas, A. Schmidt, A. Lioy, G. Russello, and F. Luccio, Eds. Springer Berlin Heidelberg, 2012, vol. 94, pp. 40–51.
9. A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "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*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 274–277.
10. A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 627–638.
11. S. Rosen, Z. Qian, and Z. M. Mao, "AppProfiler: a flexible method of exposing privacy-related behavior in android applications to end users," in *Proceedings of the third ACM conference on Data and application security and privacy*, ser. CODASPY '13. New York, NY, USA: ACM, 2013, p. 221232.
12. P. Berthome, T. Fecherolle, N. Guilloteau, and J. F. Lalande, "Repackaging android applications for auditing access to private data," in *Availability, Reliability and Security (ARES), 2012 Seventh International Conference on*, 2012, pp. 388–396.
13. "Permission explorer." [Online]. Available: https://play.google.com/store/apps/details?id=com.carlocriniti.android.permission_explorer&hl=en
14. "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
15. A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: converting android dalvik bytecode to jimple for static analysis with soot," in *Proceedings of the ACM SIGPLAN In-*

ternational Workshop on State of the Art in Java Program analysis, ser. SOAP '12. New York, NY, USA: ACM, 2012, pp. 27–38.
16. R. Vallee-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a java bytecode optimization framework," 1999.
17. "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/
18. H. Stephan and R. Siegfried, "Javascript in android apps an attack vector," 2013. [Online]. Available: http://www.bodden.de/2013/09/16/javascript-in-android-apps-an-attack-vector/
19. "Dalvik debug monitor server." [Online]. Available: http://developer.android.com/tools/debugging/ddms.html
20. "Android systrace tool." [Online]. Available: http://developer.android.com/tools/help/systrace.html
21. "Jad java decompiler." [Online]. Available: http://varaneckas.com/jad/
22. "ASM - home page." [Online]. Available: http://asm.ow2.org/
23. "A tool for reverse engineering android apk files." [Online]. Available: https://code.google.com/p/android-apktool/
24. "androguard - reverse engineering, malware and goodware analysis of android applications and more (ninja !) - google project hosting." [Online]. Available: http://code.google.com/p/androguard/
25. "ModifyApkWithDexTool - 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
26. "Droidbox, android application sandbox." [Online]. Available: http://code.google.com/p/droidbox/
27. "smali, an assembler/disassembler for android's dex format." [Online]. Available: http://code.google.com/p/smali/
28. "Jasmin, an assembler for the java virtual machine." [Online]. Available: http://jasmin.sourceforge.net/
29. R. Vallee-Rai and L. J. Hendren, "Jimple: Simplifying java bytecode for analyses and transformations," 1998.
30. K. Ali and O. Lhoták, "Application-only call graph construction," in *Proceedings of the 26th European Conference on Object-Oriented Programming*, ser. ECOOP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 688–712. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31057-7_30
31. I. Google, "Andoid operating system." [Online]. Available: http://source.android.com/
32. I. Apple, "ios." [Online]. Available: http://www.apple.com/ios/
33. D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "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*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 73–84.
34. K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "Pscout: analyzing the android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 217–228.
35. A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google android: A comprehensive security assessment," *Security Privacy, IEEE*, vol. 8, no. 2, pp. 35–44, 2010.
36. V. Rastogi, Y. Chen, and W. Enck, "Appsplayground: automatic security analysis of smartphone applications," in *Proceedings of the third ACM conference on Data and application security and privacy*, ser. CODASPY '13. New York, NY, USA: ACM, 2013, pp. 209–220.
37. A. R. Beresford, A. Rice, N. Skehin, and R. Sohan, "Mockdroid: trading privacy for application functionality on smartphones," in *Proceedings of the 12th Workshop on Mobile*

Computing Systems and Applications, ser. HotMobile '11. New York, NY, USA: ACM, 2011, pp. 49–54.
38. D. Schreckling, J. Kstler, and M. Schaff, "Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for android," *Information Security Technical Report*, vol. 17, no. 3, pp. 71–80, feb 2013.
39. P. Kodeswaran, V. Nandakumar, S. Kapoor, P. Kamaraju, A. Joshi, and S. Mukherjea, "Securing enterprise data on smartphones using run time information flow control," in *Proceedings of the 2012 IEEE 13th International Conference on Mobile Data Management (mdm 2012)*, ser. MDM '12. Washington, DC, USA: IEEE Computer Society, 2012, p. 300305.
40. D. Feth and A. Pretschner, "Flexible data-driven security for android," in *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*, ser. SERE '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 41–50.
41. T. Werthmann, R. Hund, L. Davi, A.-R. Sadeghi, and T. Holz, "Psios: bring your own privacy & security to ios devices," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, ser. ASIA CCS '13. New York, NY, USA: ACM, 2013, pp. 13–24.
42. M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "PiOS: Detecting Privacy Leaks in iOS Applications," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2011.
43. X. Xiao, N. Tillmann, M. Fahndrich, J. D. Halleux, and M. Moskal, "User-aware privacy control via extended static-information-flow analysis," *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, vol. 0, pp. 80–89, 2012.
44. A. P. Fuchs, A. Chaudhuri, and J. S. Foster, "Scandroid: Automated security certification of android applications," Tech. Rep., 2009.
45. A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: user attention, comprehension, and behavior," in *Proceedings of the Eighth Symposium on Usable Privacy and Security*, ser. SOUPS '12. New York, NY, USA: ACM, 2012, pp. 3:1–3:14.
46. M. Bishop, *Computer Security: Art and Science*, 1st ed. Addison-Wesley Professional, dec 2002.

European Commission
EUR 26484 EN – Joint Research Centre – Institute for the Protection and Security of the Citizen

Title: Mobile Applications Privacy

Authors: Igor NAI FOVINO, Ricardo NEISSE, Dimitris GENEIATAKIS, Ioannis KOUNELIS

Luxembourg: Publications Office of the European Union

2014 – 31 pp. – 21.0 x 29.7 cm

EUR – Scientific and Technical Research series – ISSN 1831-9424

ISBN 978-92-79-35409-0

doi:10.2788/66345

LB-NA-26484-EN-N

JRC Mission

As the Commission's in-house science service, the Joint Research Centre's mission is to provide EU policies with independent, evidence-based scientific and technical support throughout the whole policy cycle.

Working in close cooperation with policy Directorates-General, the JRC addresses key societal challenges while stimulating innovation through developing new methods, tools and standards, and sharing its know-how with the Member States, the scientific community and international partners.

Serving society
Stimulating innovation
Supporting legislation

doi:10.2788/66345

ISBN: 978-92-79-35409-0

Publications Office