A New Protection for Android Applications

ER-RAJY Latifa, EL KIRAM My Ahmed

Faculty of Sciences Semlalia, University Cadi Ayyad, Marrakech, Morocco

***Abstract* — Today, Smartphones are very powerful, and many of its applications use wireless multimedia communications. Prevention from the external dangers (threats) has become a big concern for the experts these days. Android security has become a very important issue because of the free application it provides and the feature which make it very easy for anyone to develop and published it on Play store. Some work has already been done on the android security model, including several analyses of the model and frameworks aimed at enforcing security standards. In this article, we introduce a tool called PermisSecure that is able to perform both static and dynamic analysis on Android programs to automatically detect suspicious applications that request unnecessary or dangerous permissions.**

***Keywords* — Android, Operating System Mobile, Privacy, Issues, Hackers, Permission, Solution, Protection.**

I. Introduction

Since the first Android powered phone named The HTC Dream (also known as the T-Mobile G1 in the United States and parts of Europe, and as the Era G1 in Poland) was delivered in October 2008 [1], Android smart phones have grown to the largest global market share (76%) among all smartphones shipped in the 4th quarter of 2014 [2]. In 2014, Google announced that more than billion Android devices had been activated [3]. This important popularity of Android and the open nature of its application marketplace makes it a prime target for attackers. Malware authors can be freely upload malicious applications to the Android Play Store waiting for unsuspecting users to download and install them.

To alert users to the privacy and security ramifications of installing an application, Android employs mandatory access control (MAC) in the form of an install-time permission system [4]. At installation time, an application must request permission to access system resources such as location, Internet, or the cellular network, from the user. Then the user is presented with a screen allowing him to either grant all the permissions or cancel the installation. Since it is not possible to selectively accept or deny access privileges. Thus, many users simply accept such permission requests without considering their implications, which put their private data in the zone of danger [5]. For example if an application granted some critical permissions such as the INTERNET permission it can controls communication with remote servers and if this application was also granted access to the Android camera, nothing prevents it from sending the user's pictures to any server on the internet. May 2014, Google had done a Play Store updates to simplify the display of the permissions and allow better navigation user. They were re-grouped by categories. Therefore, from more than 150 permissions, we went to a dozen groups, including another category, which includes everything that does not fit elsewhere [6]. With the old system, each update of the application, if developer added a new permission, the Play Store posted it and the user must then accept it. With the new system, developers can add for example the permission ACCESS_SUPERUSER that allow him to take control of all the features of the phone and storage if his application had permissions in the group "another category" [7].

The rest of the paper is organized as follows. In Section 2, we give an overview on the background of our work. Section 3 describes the details on the problem caused by permissions requested by android applications and their updates. Section 4 gives an overview on related work. We proceed in Section 5 with details of the design and user interface of our tool. In Section 6, we demonstrate how our tool protects the user from permissions and updates challenges by providing the results of applying PermisSecure to 120 paid and 456 free applications from the Google Play Store. In Section 7 we conclude and summarize our results and contributions.

II. Background

To frame the problem, we describe the Android architecture, permission system and explain how these permissions requested by an application can play an important role in spring of malwares.

A. Application Package

An Android application package or apk file is an archive. It contains a Dalvik executable (dex file), which is a compiled Android program that runs on a Dalvik virtual machine, and a set of resources (non-executable files like graphics, media files, user interface components, etc.). Application packages also contain a manifest (AndroidManifest.xml), which Android contains meta-information about the application like package name, version, supported Android versions, and other attributes. These components are digitally signed with the developer's signing key. The developer's signing certificate can be self-signed and is included in the application package [8].

B. Deconstructing Application Installation

Any developer (even those who have not registered with Google) can create and distribute applications through the official Google Play Store, through third-party markets (e.g., Amazon Appstore) or through developer websites (side-loading). The lack of control over the application distribution process raises the importance of enforcing security within the Android OS. During the installation of a new application, permissions are approved prior to installation, but the rest of the process remains the same. First, the application package validity is verified: the system ensures that the Android application package has not been modified or corrupted since being signed, and that it contains a valid certificate for the signing key. Then, Android decides whether the application is a new installation or should overwrite an existing application. If the application being installed has the same package attribute in the manifest (e.g., com.google.android.music) as another currently installed application, then Android will treat the installation as an update. So, the certificate (or set of certificates if signed by multiple keys) is compared to the certificate(s) of the already installed application. If both applications were signed with the same key(s), then the currently installed binary is removed (preserving any user data) and the new application is installed in its place. Otherwise, the new application is installed as an initial installation. Next Android must assign a UID to the application. In this case, the previous application's UID is used. In the case of an initial installation, Android checks if the application's manifest contains the sharedUserId directive. If so,

DOI: 10.9781/ijimai.2016.372

Android looks for any other installed applications that are signed with the same key(s) and have sharedUserId specified in their manifest. If such applications are found, the application is assigned the same UID; otherwise a new UID is created.

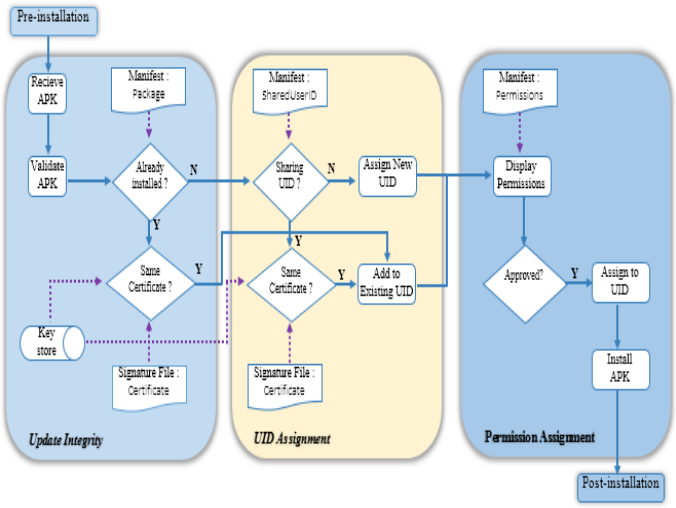


Fig 1. Abstract model of the Android installation process for an application package (apk).

Finally permissions must be assigned to the UID[9]. The user is prompted to review and approve the permission assignments before the application is installed. When UID sharing is not used, permissions listed in the application's manifest are assigned to the UID. When UID sharing is used, the UID is assigned the union of all permissions in the manifests of applications sharing the UID. If the application is updating an already installed application, the permissions listed in the updated application's manifest are assigned to the UID [10].

III. Problem Description

Permission System: Android controls access to system resources with install-time permissions by using two-way process. Firstly, developer defines the required permissions that are the first requisite for performing the functionalities of an application. Secondly, during installation time, user must have approved all the requested permissions to use an application [11]. Permissions requested by each application permissions fall into four levels:

(1) Normal – These permissions protect access to API calls that cannot impart real harm to the user (e.g. SET_WALLPAPER controls the ability to change the user's background wallpaper) and, while applications need to request them, they are automatically granted.

(2) Dangerous – These control access to potentially harmful API calls, like those related to spending money or gathering private information. For example, Dangerous permissions are required to send text messages, read the list of contacts, call numbers, open Internet connections, etc.

(3) Signature – These regulate access to the most dangerous privileges, such as the ability to control the backup process or delete application packages. They are automatically granted to a requesting application if that application is signed by the same certificate (so, developed by the same entity) as that which declared/created the permission. This level is designed to allow applications that are part of a suite, or otherwise related, to share data.

(4) Signature/System – Same as Signature, except that the system image gets the permissions automatically as well. This is designed for use by device manufacturers only.

Menace of Pileup: The security analysis of mobile updating, focusing on Android Package Manager brings to light a new category of unexpected and security-critical vulnerabilities within Android's update installation logic. Such vulnerabilities, called Pileup (privilege escalation through updating), enable an unprivileged malicious application to acquire system capabilities once the OS is upgraded, without being noticed by the phone user. More specifically, through the application running on a lower version Android, the adversary can strategically claim a set of carefully selected privileges or attributes only available on the higher OS version. For example, the application can define a new system permission such as permission. ADD_VOICEMAIL on Android 2.3.6, which is to be added on 4.0.4. It can also use the shared user ID[12] (a string specified within an application's manifest file) of a new system application on 4.0.4, its package name and other attributes. Since these privileges and attributes do not exist in the old system (2.3.6 in the example), the malicious application can silently acquire them (self-defined permission, shared UID and package name, etc.). When the system is being updated to the new one, the Pileup flaws within the new Package Manager will be automatically exploited. Consequently, such an application can stealthily obtain related system privileges, resources or capabilities. In the above example, once the phone is upgraded to 4.0.4, the application immediately gets permission.ADD_VOICEMAIL without the user's consent and even becomes its owner, capable of setting its protection level and description. Also, the preempted shared UID enables the malicious application to substitute for system applications such as Google Calendar, and the package name trick was found to work on the Android browser, allowing the malicious application to contaminate its cookies, cache, security configurations and bookmarks, etc.

IV. Related Work

Research related to this work can be classified into the following categories:

Android Permissions: Previous studies of Android applications have been limited in their understanding of permission usage. Enck et al. apply Fortify's Java static analysis tool to decompiled applications; they analyze a large set of applications and study their API use [13]. However, they are limited to studying application's use of a small number of permissions and API calls. In a recent study, Felt et al. manually classify a small set of Android applications as over privileged or not [14]. They were unable to reliably differentiate between necessary and unnecessary permissions because of limited Android documentation.

Update behavior of Android: The first studies has been by done Moeller et al. [15] by analyzing the updates of applications from Google Play quantitatively. An attack of the android system called Application Update Attack is studied by Tenenboim et al. [16]. Application updates might be a potential way to implant new security vulnerability and privacy data leaks to the user's phones. Chin et al. [17] studied users's confidence in security and privacy of Android. They found that users reported various concerns because of some misconceptions or misunderstandings. Android permissions provide a mechanism for users to manage the access control of applications, especially when fine-grained controls are granted.

Privilege escalation: XManDroid [18] was designed to prevent privilege escalation and collusion attacks by enforcing policies on the communications between applications, e.g., banning an application that has access to the user's location from interacting with an application that is allowed to access the Internet. Aurasium [19] repackages applications and introduces an intermediate layer between the framework's native code libraries and the operating system kernel inside the application process. AppSealer [20] combines static- and

dynamic-code analysis techniques to patch the applications's bytecode in order to mitigate component hijacking attacks. Xing et al.[7] Authors introduce a scanner tool for detecting applications that are vulnerable against the pileup threat.

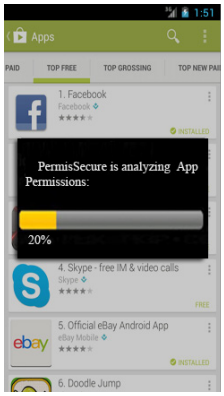
V. PermisSecure Introduction

In this section, we present PermisSecure, a tool sandbox based on dynamic and static analysis for Android smartphones, which increases user awareness about potentially harmful applications in install-time on his phone. The concept is based on the assumption that users are willing to take security increasing actions (such as canceling the installation of a potentially harmful application) once they gain knowledge about a potential dangerous or unnecessary permissions requested.

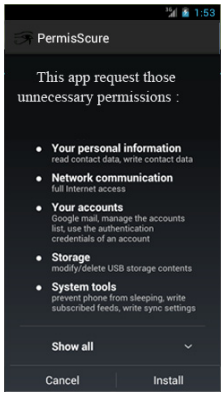
A. Interface User

The design and implementation of the PermisSecure user interface follows the basic principle of warning, to provide the user with only as many pieces of information as necessary to avoid suspicious applications. Further, we designed the application to integrate seamlessly in the Android system.

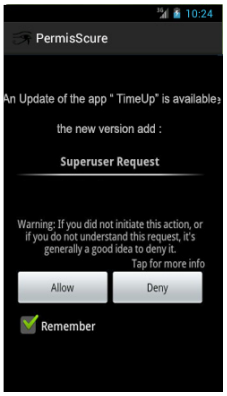
When the user wants to install a new application, the software involved to perform an analysis on two levels (See Fig. 5(a)) , the first level it analyzes the permissions requested by the application, if they are dangerous it displays a warning to the user who can choose between contain or cancel the installation (See Fig. 5(b)). Otherwise, it goes to the second level of analysis and verifies the code. Therefore, if the code is suspicious, it shows to the user a warning same to that of the first level, otherwise it allows the installation of the application. PermisSecure repeat the process when an application update is available (See Fig. 5(c)).



(a) PermisSecure preform an analysis in install-time of new application.



(b) PermisSecure analyzes permissions and displays a warning to the user who can choose between contain or cancel the installation.



(c) Once an application is available, PermisSecure analyzes it.

VI. Results

We applied PermisSecure to 120 paid and 456 free applications from the Google Play Store. For the applications to identify the dangerous or unnecessary permissions. PermisSecure calculates the maximum set of Android permissions that an application may need. We compare that set to the permissions actually requested by the application. If the application requests more permissions, then it is over privileged.

Unnecessary permission: PermisSecure identified that 45% of free and 22% of paid applications have unnecessary permissions. In some cases, we were able to determine why developers asked for unnecessary permissions.

- Permission Name: Developers sometimes request permissions with names that sound related to their application's functionality, even if the permissions are not required.
- RelatedMethods: Some classes contain a mix of permission-protected and unprotected methods. We have observed applications using unprotected methods but requesting permissions that are required for other methods in the same class. For example, android.provider.Settings.Secure includes both setters and getters. Setters require the WRITE_SETTINGS permission, and the getters do not. Two applications use the getters and not the setters, but request the WRITE_SETTINGS permission.
- Copy and Paste: Popular message boards contain Android code snippets and advice about permission requirements. Sometimes this information is inaccurate, and developers who copy it will over privilege their applications. For example, one application in our data sets registers to receive the android.net.wifi.STATE_CHANGE Intent and requests the ACCESS_WIFI_STATE permission. As of May 2011, the third-highest Google search result for that Intent contains the incorrect assertion that it requires that permission [21].
- Deputies: An application can send an Intent to another deputy application, asking the deputy to perform an operation. If the deputy makes a permission-protected API call, then the deputy needs the permission. The sender of the Intent, however, does not. We noticed instances of applications requesting permissions for actions that they asked deputies to do. For example, one application asks the Android Market to install another application. The sender asks for INSTALL_PACKAGES, which it does not need because the Market application does the installation. Another application asks the built-in camera application to take photos, yet requests the CAMERA permission for itself.
- Testing Artifacts: A developer might add a permission during testing and then forget to remove it when the test code is removed. For example, ACCESS_MOCK_LOCATION is typically used only for testing but can be seen in released applications.

Confusion over permission names, related methods, and Intents could be addressed with improved API documentation. We recommend listing permission requirements on a per-method (rather than per-class) basis. Confusion over deputies could be reduced by clarifying the relationship between permissions and pre-installed system applications.

Permission	*Types*		*Permission level*
	Paid	*Free*	
ACCESS_NETWORK_STATE	10%	15%	Normal
READ_PHONE_STATE	9%	15%	Dangerous
ACCESS_WIFI_STATE	6%	8%	Normal
WAKE_LOCK	3%	5%	Dangerous
WRITE_EXTERNAL_STORAGE	5%	7%	Dangerous
ACCESS_MOCK_LOCATION	4%	6%	Dangerous
CALL_PHONE	4%	5%	Dangerous
ACCESS_COARSE_LOCATION	3%	5%	Dangerous
CAMERA	2%	5%	Dangerous
INTERNET	3%	5%	Dangerous

This table shows that almost all unnecessary requested by applications (paid and free) are dangerous.

Dangerous permissions: We are concerned with the prevalence of dangerous permissions. Dangerous permissions are displayed as

a warning to users during installation and can have serious security ramifications if abused. We find that 93% of free and 82% of paid applications have at least one dangerous permission, e.g., generate at least one warning. Android permissions are grouped into functionality categories. This provides a relative measure of which parts of the protected API are used by applications. A small number of permissions are requested very frequently. In particular, the INTERNET permission is heavily used. We find that 14% of free and 4% of paid applications request INTERNET as their only dangerous permission. Barrera et al. hypothesize that free applications often need the INTERNET permission only to load advertisements [22]. The disparity in INTERNET use between free and paid applications supports this hypothesis, although it is still the most popular permission for paid applications. Enck et al. found that some free applications leak personal data [23][24], this may explain the difference in ACCESS COARSE LOCATION requests. The prevalence of the INTERNET permission means that most applications with access to personal information also have the ability to leak it. For example, 97% of the 225 applications that ask for ACCESS FINE LOCATION also request the INTERNET permission. Similarly, we find that 99%, 94%, and 78% of the 306, 149, and 14 respectively applications that request ACCESS COARSE LOCATION, READ CONTACTS, and READ CALENDAR have the INTERNET permission. Although many applications ask for at least one Dangerous permission, the total number of permission requests is typically low. Even the most highly privileged application in our set asks for less than half of the available 56 dangerous permissions. Fig. 3 shows the distribution of dangerous permission requests.

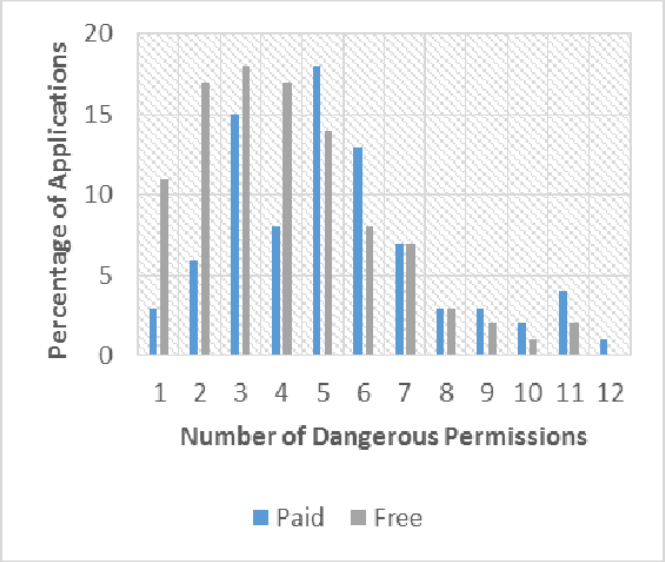


Fig 3. Dangerous permissions per application

Analysis of updates: The success of a privilege escalation attack on an update process depends not only on the presence of Pileup vulnerabilities, but also on the new system resources and capabilities the update adds that can be acquired by the adversary through the attack. Here we present a measurement study in which we ran our PermisSecure against a large number of applications for updates (183 Apps) to understand the exploit opportunities (new exploitable attributes and properties) they bring in.

We first looked at the overall impacts of the Pileup vulnerabilities to the Android ecosystem, in terms of update instance, which refers to the upgrade of a specific OS (from a specific manufacturer, on a specific device model and for a specific carrier) to a higher one under the same set of constraints. For each update instance, we measured the quantity of exploiting opportunities it can offer, with regards to all the Pileup flaws found in our research, such as the numbers of new permissions, packages and shared UIDs an update instance introduces to the new system. From the 383 we downloaded, we identified 241 update instances. The Statistics on their total exploit opportunities in each instance are illustrated in Fig. 4. Particularly, we found that 50% of those instances have more than 71 opportunities.

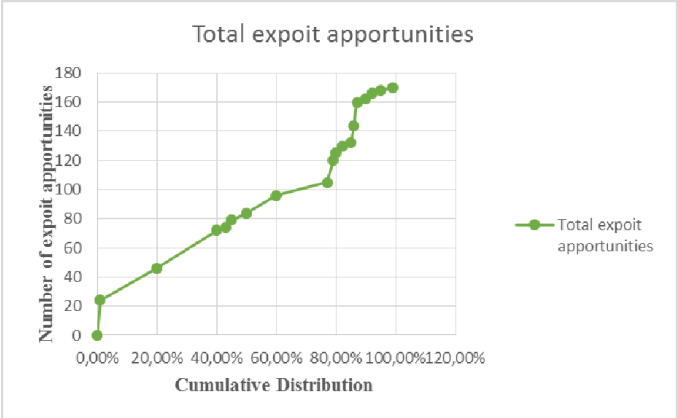


Fig 4. Cumulative Distribution of Total Exploit Opportunities in Each Update Instance.

VII. Conclusion and Future work

In this paper, we present the design and implementation of PermisSecure that analyzes permissions requested by Android applications in installing-time and after their updates. Our reference implementation is very efficient and induces a small performance overhead. Therefore, we have developed this tool especially for users without a technical and security background. Our aim was to put permission based systems on a stable footing by informing users about dubious permission sets. There are several ways to extend the concept of creating awareness after applications are installed. First, it would be beneficial if the user would be informed before installation for suspicious applications. Preferably, the user should be given alternatives, such as this torch light app needs 124 permissions, including 15 dangerous ones. Alternatively, we have found a torch light application, that only needs 2 permissions, none of them being dangerous.

References

[1] M. Amir, "Energy-Aware Location Provider for the Android Platform," University of Alexandria, 2010.

[2] Scientiamobile, "Mobile Overview Report October - December 2014," Reston, 2014.

[3] L. Li, A. Bartel, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis," CoRR, 2014.

[4] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android Permissions : User Attention , Comprehension , and Behavior," 2012.

[5] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, "L4Android: A Generic Operating System Framework for Secure Smartphones," Proc. 1st ACM Work. Secur. Priv. Smartphones Mob. Devices, pp. 39–50, 2011.

[6] N. Viennot, E. Garcia, and J. Nieh, "A measurement study of google play," 2014 ACM Int. Conf. Meas. Model. Comput. Syst. - SIGMETRICS '14, pp. 221–233, 2014.

[7] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, "Upgrading Your Android, Elevating My Malware: Privilege Escalation Through Mobile OS Updating," IEEE Symp. Secur. Priv., 2014.

[8] T. Report, "Analysis of Dalvik Virtual Machine and Class Path Library,"

Management, pp. 1 – 42, 2009.

[9] E. Struse, J. Seifert, S. Ullenbeck, E. Rukzio, and C. Wolf, "PermissionWatcher: Creating user awareness of application permissions in mobile systems," Third Int. Jt. Conf. Ambient Intell., pp. 65–80, 2012.

[10] D. Barrera, "Securing Decentralized Software Installation and Updates," 2014.

[11] T. Vidas, N. Christin, and L. F. Cranor, "Curbing Android Permission Creep," IEEE Web 2.0 Secur. Priv. Work., 2011.

[12] L. Davi, A. Dmitrienko, A. R. Sadeghi, and M. Winandy, "Privilege escalation attacks on android," Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics), vol. 6531 LNCS, pp. 346–360, 2011.

[13] W. Enck, D. Octeau, P. Mcdaniel, and S. Chaudhuri, "A Study of Android Application Security."

[14] A. Felt, K. Greenwood, and D. Wagner, "The effectiveness of application permissions," WebApps '11 2nd USENIX Conf. Web Appl. Dev., pp. 75–86, 2011.

[15] A. Möller, F. Michahelles, S. Diewald, L. Roalter, and M. Kranz, "Update Behavior in App Markets and Security Implications : A Case Study in Google Play," Proc. 3rd Int. Work. Res. Large. Held Conjunction with Mob. HCI, pp. 3–6, 2012.

[16] O. Barad, a Shabtai, D. Mimran, L. Rokach, B. Shapira, and Y. Elovici, "Detecting Application Update Attack on Mobile Devices through Network Features," pp. 2465–2466, 2013.

[17] E. Chin, A. P. Felt, V. Sekar, and D. Wagner, "Measuring user confidence in smartphone security and privacy," Proc. Eighth Symp. Usable Priv. Secur. - SOUPS '12, no. 1, p. 1, 2012.

[18] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A. Sadeghi, "XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks," System, pp. 4–7, 2011.

[19] R. Xu, H. Saïdi, R. Anderson, and H. Saıdi, "Aurasium: Practical Policy Enforcement for Android Applications," Proc. 21st USENIX Conf. …, p. 27, 2012.

[20] M. Zhang and H. Yin, "AppSealer: Automatic Generation of Vulnerability-Specific Patches for Preventing Component Hijacking Attacks in Android Applications," Symp. Netw. Distrib. Syst. Secur., no. February, pp. 23–26, 2014.

[21] Jacob West, "Software Security Goes Mobile," AppSecAsiaPac2012, 2012.

[22] D. Barrera, P. C. Van Oorschot, and A. Somayaji, "A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android," Security, no. 1, pp. 73–84, 2010.

[23] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones.," Osdi '10, vol. 49, pp. 1–6, 2010.

[24] J. Pascual-Espada et al., "Using extended web technologies to develop Bluetooth multi-platform mobile applications for interact with smart things", Information Fusion, vol. 21, pp. 30-41, 2015

ER-RAJY Latifa received the B. S. in administration of computer systems from faculty of Sciences, University Mohamed V at Rabat in 2011. She obtained her master in the field of networks and computer system from faculty of sciences and techniques, University 1st Hassan at Settat in 2013. She is currently a Ph.D student in the University Cadi Ayyad, Marrakech, Morocco. Her main field of research interest is the security of android application.

Moulay Ahmed El Kiram is research professor at the Faculty of Science Semlalia, Cadi Ayyad University of Marrakech. He received his DES in Computer Science in 1997 at Mohammed V University of Rabat. El Kiram specializes in Security and network communication. His areas of interest include Authentication, particularly in multicast environment.