International Journal of Emerging Technology in Computer Science & Electronics (IJETCSE)
ISSN: 0976-1353 Volume 24 Issue 4 – MARCH 2017.

# REASSESSMENT OF PERMISSION GAPS IN ANDROID APPLICATIONS

V.R.Niveditha[#1] and Dr. ShobaRani[*2]

[#] *PG scholar ,M.TECH Information Security And Cyber Forensics, Dr. M.G.R. Educational and Research Institute University, Chennai, India*

[*] *Professor, Computer Science and Engineering, Dr. M.G.R. Educational and Research Institute University, Chennai, India*

***Abstract*— Android is a widely used mobile operating system among the social users. Due to its robust security framework and security at different levels of layered system, the popularity of the android OS increased. Permission framework of the android system is one of the significant features for providing access controls. Several applications are granted with a set of permissions that controls the privacy data.The application can have access to these privacy sensitive resources only when the permissions that the application asks for have been granted by the user at the time of installation of the application. In this paper, we have analyzed the permission gaps persist in android applications. It is processed out in three analyses, namely, manifest analysis, static analysis and dynamic analysis. In the manifest file, the permission of the system is enabled and collected the basic apps details of the android systems. Similarly, the static and dynamic analysis is performed from Activity Control Manager of the android system. Experimental analysis is carried out in 1055 apps of 10 categories. Results have distinguished the requested permission (PR) and user's permission (PU) of the android system.**

***Index Terms*— Android, Permission framework, Operating system, Manifest analysis, and ActivityControlManager**

## I. INTRODUCTION

The security architecture of the mobile operating systems Android and Blackberry as well as other systems such as the Google Chrome browser extension system, use a similar security model called the permission-based security model [1]. A permission based security model can be loosely defined as a model in which 1) each application is associated with a set of permissions that allows accessing certain resources1; 2) permissions are explicitly accepted by users during the installation process and 3) permissions are checked at runtime when resources are requested.In Android, the authorization model is embedded into the "Android framework". The framework exposes an Application Programming Interface (API) that contains classes and methods for developers to interact with the system resources. For instance, the API contains a method getGPSLocation2 which gives the current GPS location of the smartphone, if available. This API method, and many others, is sensitive with respect to security or privacy. Consequently, in response to a call to getGPSLocation, the framework checks that the caller has been explicitly granted the GPS permission [2]. This permission model has an impact on the development process of applications. To write an application, developers must identify, for each API method they use, the permissions that must be declared for the application to work correctly.

They need a mapping between the API methods and the required permissions. In the case of Android, the mapping is given by the official documentation. However, the documentation is not always up-to-date or clear and, consequently, question-and-answers website are full of questions regarding the use of permissions [3]. As a result, inventors often either underor over-estimate the required permissions. Missing permission causes the application to crash. Adding too many of them is not protected. In the latter case, inserted malware can use those declared, yet unused permissions, to achieve malicious goals. We call those unused permissions, "permission gap". Any permission gap results in insecure, suspicious or unreliable applications. It enables developers to easily declare the permissions they actually need: not more, not less. To extract this map, we explore in this paper the use of static examination to extract the permission checks. On a framework of the scale and sophistication of Android, naive approaches using off-the-shelf static examination fail miserably.

Permission-based software is conceptually divided in three layers [4]: 1) the core platform (the operating system) which is able to access all system resources (e.g., change the network policy); 2) a middleware answerable for providing a clean application programming interface (API) to the OS resources and for checking that applications have the correct permissions when they want accessing them; 3) applications constructed on top of the middleware. They have to explicitly declare the permissions they require. Layers #2 and #3 motivate the generic label "permission-based software". Since the middleware also hides the OS difficulty and provides an API, it is sometimes called, as in the case of Android, a "framework" [5].

The rest of the paper is organized as follows: Section II depicts the related work; Section III depicts the proposed work; Section IV depicts experimental analysis and concludes in Section V.

## II. RELATED WORK

Similar to the improvement in the desktop PC world, the

early systems for analysis of Android apps used a static approach. A typical system for this approach was proposed by [6]. They attempt to extract the function calls from an Android application [8](using the readelf utility) and compare the resulting list with the data of known malware [7].

Another example for the static approach is Androguard by [8,10], which decompiles the application and applies signature-based malware detection. This system is completely open source. In response to static examination systems in the desktop PC world, malware authors established various obfuscation techniques that have been shown to be effective against static examination [11]. This is also an emerging trend in Android applications, and it is clear that static examination alone cannot ensure complete analysis coverage anymore. Therefore, researchers have begun to develop systems for dynamic analysis of Android apps.

One of the first such systems is TaintDroid by [12]. It is an efficient and dynamic taint-tracking system that provides real-time analysis by leveraging Android's execution atmosphere. This system was complemented with a fully automated user emulation and reporting system by [13] and is available under the name Droidbox. Droidbox is an active tool to analyze Android apps; however, it absences support to track native API calls. In fact, we are unaware of any system that supports native API call tracking during dynamic analysis to date. TaintDroid and Droidbox are open source and publicly available [14].

Another interesting system using dynamic analysis is pBMDS by [15]. It uses machine learning to create user and system outlines for a specified behavior. Afterward, it tries to correlate user inputs with system calls by comparing their behavior profiles to detect anomalous application activities. This system was built for Symbian OS and verified with a very small dataset. CrowDroid, by Burguera [16] uses a similar approach, but with a much wider set of behavior data and with a more advanced monitoring system. CrowDroid uses strace, a debugging value for Linux and some other Unix-like systems, to monitor every system call and the signals it receives. CrowDroid, however, does not consider information from Android's Dalvik VM [17].

The system AASandbox of [18] was the first system combining static and dynamic analysis in a very basic way for the Android platform. Unfortunately, AASandbox does not look to be maintained anymore. Another system combining static and dynamic analysis is DroidRanger by Zhou et al. [19]. DroidRanger implements a combination of permission-based behavioral footprinting to identify samples of already known malware families and a heuristicbased filtering scheme to detect unknown malicious families. With this approach, they were able to detect 32 malevolent samples inside the official Android Market in June 2011. Within their dynamic part, they use a kernel module to log only system calls used by known Android exploits or malware.

The system that is most alike to our approach is Andrubis from the Vienna University of Technology. In their approach, they also use Droidbox and TaintDroid for automatic analysis, but they are limited to applications that need a minimum of Android 2.3 to be able to run on a device (which is equal to API level 9 as can be seen in the platform versions overview of Android [20]). In contrast, we are able to analyze applications that support a minimum API level of 13 and will be able to analyze applications that are build for API level 17 and under by end of March 2014. This difference can be very significant when you compare the market share of API level 7 and below (1.3%) to the share of API level 17 and lower (75.4%). Additionally, they are not able to track calls confidential native code at the time of this writing.

## III. PROPOSED WORK

This section depicts the working of enhanced permission based android system. The permission analysis over android app is given in three levels: a) Manifest analysis b) static code analysis and c) Dynamic analysis. The target of the study is to estimate the difference between requested permissions (PR) and dynamic analysis of the used permissions (PU).

### *A. Manifest analysis:*

In order to access any files, the developers should given permission in manifest files.We can found the code <uses-permission >elements in manifest file. By doing so, we can access the request permissions.

### *B. Static analysis:*

There are three types of permission based operations are given as i) Functional calls that lead to check permission method b) Sending and receiving methods for intents c) techniques for involving the content providers. In order to derive which permissions are used, the similar method STOWAWAY is used for measuring the used permissions. A lightweight dissembler is used for DEX bytecode on API invocations, intents, and content providers from an app. Similarly, the classes that inherits from the app-defined methods and android defined methods. Then, the permission related information of API (PUs-a), intents (PUs-i) and content providers (PUs-c) of the permission gap is given as:

$PU_s = PU_{s-a} \cup PU_{s-i} \cup PU_{s-c}$

### *C. Dynamic analysis:*

In the dynamic analysis, java reflection and dynamic loading are used for analysing the permissions in android system. The ActivityManagerService contain *check permission* method which is used for enabling the permissions. 107 permissions are used for protecting the files in the android systems. It is also adequate to capture the permissions used by the analyzed apps. Monkey is a command-line tool that runs on an emulator or device, which can generate pseudo-random sequences of user events such as clicks, touches, and gestures, as well as a number of system-level events. Note that although dynamic analysis cannot catch all the permissions due to the difficulty in achieving full coverage of the whole execution path, it works as a useful complement to static analysis because it can find more permission related to dynamic behaviors

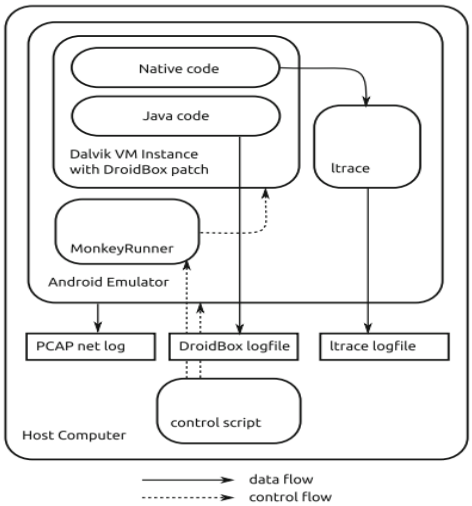


Fig.1. Dynamic analysis of the android component

## IV. EXPERIMENTAL RESULTS

An experimental analysis is carried out over 1, 055 apps of 10 categories from Google Play. Both static and dynamic analysis is carried out for each app. We dynamically write the scripts to test the apps. The table 1 describes the results of permission analysis.

TABLE 1. RESULTS OF PERMISSION ANALYSIS

| Category | No.of apps | Avg. of PRs | Avg of PUs |
|---|---|---|---|
| Wallpaper | 107 | 5.69 | 5.33 |
| Widgets | 103 | 10.36 | 9.25 |
| Books | 105 | 6.8 | 6.23 |
| Business | 105 | 3.14 | 4.63 |
| Comics | 109 | 6.50 | 7.11 |
| Finance | 105 | 7.61 | 8.14 |
| Tools | 105 | 9.89 | 7.78 |

TABLE 2. NO. OF APPS ANALYSIS BETWEEN DYNAMIC AND STATIC ANALYSIS

| Category | No.of apps with new permissions | % of apps | Avg. no. of new permissions |
|---|---|---|---|
| Wallpaper | 38 | 36.3% | 2.36 |
| Widgets | 61 | 53.2% | 2.96 |
| Books | 53 | 55.3% | 1.98 |
| Business | 51 | 73.3% | 1.86 |
| Comics | 88 | 61.2% | 2.01 |
| Finance | 74 | 72.5% | 2.69 |
| Tools | 76 | 76.3% | 2.31 |

The comparison between dynamic analysis and static analysis of the apps with permissions to other apps are examined. We find that the percentage of these apps using dynamic behaviors is higher than the other apps, which indicates that dynamic analysis is more effective in handling dynamic behaviors (such as Java Reflection and Dynamic Loading) than static analysis. Similarly, we also analyzed the overprivilege permission analysis of the android apps. The results also indicate that if we use only static analysis to detect permission overprivilege, we might not be able to identify all the permissions used, which might result in high false positives in permission overprivilege detection. If we combine dynamic analysis and static analysis, we are able to eliminate more than 8% of those apps that are categorized as overprivileged.

## V. CONCLUSION

This paper concentrated on the effectiveness of Android permissions. Our studies represented those permissions of the android fails in majority of the users.A minority of users demonstrated alertness and understanding of permissions, and we found that permissions helped some users avoid privacy-invasive applications. This motivates continued effort towards the goal of usable permissions. At first, we examined about the set of issues in android permissions in three analyses, namely, manifest analysis, static analysis and dynamic analysis. We provide a set of recommendations to address these issues. Our results also support three directions of future work for improving permission systems: connecting reviews to permissions, customizing warnings to users' concerns, and investigating new types of warning dialogs.

## REFERENCES

[1] Sophos mobile security threat report 2014. Available online: http://www.sophos.com/en-us/medialibrary/PDFs/other/sophos-mobilesecurity-threat-report.pdf?la=en [Accessed Nov. 2014]

[2] M. Turk and A. Pentland "Eigenfaces for recognition," J. cognitive Neurosci. 3, pp. 71-86, 1991

[3] A. Apvrille and T. Strazzere, "Reducing the window of opportunity for Android malware Gotta catch 'em all," Journal in Computer Virology vol. 8, No. 1-2, pp. 61-71, 2012.

[4] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. "RiskRanker: scalable and accurate zero-day android malware detection". In Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys '12). ACM, New York, NY, USA, 2012, pp. 281-294.

[5] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. "Analyzinginterapplication communication in Android" In Proc. of the 9th international conference on Mobile systems, applications, and services (MobiSys '11). ACM, New York, NY, USA, 2011, pp. 239-252.

[6] P. P.F. Chan, L. C.K. Hui, and S. M. Yiu. "DroidChecker: analyzing android applications for capability leak" In Proc. of the fifth ACM conference on Security and Privacy in Wireless and Mobile Networks (WISEC '12). ACM, New York, NY, USA, 2012, pp.125-136.

[7] W. Dong-Jie, M. Ching-Hao, W. Te-En, L. Hahn-Ming, and W. KuoPing, "DroidMat: Android malware detection through manifest and API calls tracing," in Proc. Seventh Asia Joint Conference on Information Security(Asia JCIS), 2012, pp. 62-69.

[8] S. Y. Yerima, S. Sezer, G. McWilliams, I. Muttik, (2013) "A new Android malware detection approach using Bayesian classification". Proc. 27th IEEE int. Conf. on Advanced Inf. Networking and Applications (AINA 2013), Barcelona, Spain.

[9] S. Y. Yerima, S. Sezer, G. McWilliams. "Analysis of Bayesian Classifcation Approaches for Android Malware Detection," IET Information Security, Vol 8, Issue 1, January 2014.

[10] H. Peng, C. Gates, B. Sarma, N. Li, A. Qi, R. Potharaju, C. Nita-Rotaru and I. Molloy. Using Probabilistic Generative Models For Ranking Risks of Android Apps. In Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS 2012), Oct. 2012.
[11] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedro, P. G. Bringas, G. Alvarez "PUMA: Permission usage to detect malware in Android" International Joint Conference CISIS'12-ICEUTE´12-SOCO´12 Special Sessions, in Advances in Intelligent Systems and Computing, Volume 189, pp. 289-298
[12] X. Liu and J. Liu "A two-layered permission-based Android malware detection scheme" in Proc. 2nd IEEE International Conferenece on Mobile Cloud Computing, Services and Engineering, Oxford, UK, 8-11 April 2014.
[13] J. Sahs and L. Khan "A Machine Learning Approach to Android Malware Detection" 2012 European Intelligence and Security Informatics Conference, 2012.
[14] D. Arp, M. Sprientzenbarth, M. Hubner, H. Gascon, K. Reick "DREBIN: Effective and explainable detection of Android malware in your pocket" NDSS 2014, February 2014, San Diego, CA, USA.
[15] A. Sharma, and S. K. Dash. "Mining API Calls and Permissions for Android Malware Detection." Cryptology and Network Security. Springer International Publishing, 2014. 191-205.
[16] S. Y. Yerima, S. Sezer, I. Muttik "Android malware detection using parallel machine learning classifiers" in Proc. 8th International Conference on Next Generation Mobile Applications Services and Technologies (NGMAST 2014), Oxford, UK, Sept. 2014.
[17] B. Kang, B-J. Kang, J. Kim, E. G. Im "Android malware classification method: Dalvik bytecode frequency analysis" RACS 2013, October 1-4, 2013, Montreal, QC, Canada.
[18] M. E. Saleh, A. B. Mohamed, and A. A. Nabi "Eigenviruses for metamorphic virus recognition" IET Inf. Secur., vol. 5, iss. 4, pp. 191-198, 2011.
[19] S. Deshpande, Y. Park and M. Stamp "Eigenvalue analysis for metamorphic detection" J. ComputVirol Hack Tech, October 2013.
[20] McAfee threat report June 2014. Available online: http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q1-2014.pdf [Accessed Nov. 2014]