Understanding the Service Life Cycle of Android Apps: An Exploratory Study

Kobra Khanmohammadi, Mohammad Reza Rejali, Abdelwahab Hamou-Lhadj
Software Behaviour Analysis (SBA) Research Lab,
Department of Electrical and Computer Engineering,
Concordia University, Montreal, QC, Canada
{k_khanm, mrejali, abdelw}@ece.concordia.ca

ABSTRACT
The fast growing use of the Android platform has been accompanied with an increase of malwares in Android applications. A popular way in distributing malwares in the mobile world is through repackaging legitimate apps, embedding malicious code in them, and publishing them in app stores. Therefore, examining the similarity between the behavior of malicious and normal apps can help detect malwares due to repacking. Malicious apps operate by keeping their operations invisible to the user. They also run long enough to perform their malicious tasks. One way to detect malicious apps is to examine their service life cycle. In this paper, we examine the service life cycle of apps. We extract various features of app services. We use these features to classify over 250 normal and malicious apps. Our findings show that malicious apps tend to use services to do their malicious operation and have no communication with the other components of the app, whereas the services in normal apps are usually bound to other components and send messages to notify users about the operations they perform. The results of this exploratory study can be used in the future to design techniques for detecting malicious apps using the classification of their service features.

Categories and Subject Descriptors
D.4.6 [**Security and Protection**]: Security and Protection-*Invasive software*

Keywords
Android Apps; Malware Detection; App Repackaging.

1. INTRODUCTION
The growing trend of using smart phones has motivated the development of mobile applications (apps) to serve users in a variety of areas including entertainment, communication and more critical activities such as banking. The number of apps available for downloading in leading app stores, Google play store and Apple app store, reached 1.5 million in May 2015 [16]. Apps' high popularity has set in motion a burgeoning development of malwares. McAfee lab's threat report shows that the number of the newly detected malwares exceeded 700,000 at the end of 2014. The report indicates an emergence of 387 new threats every minute, or more than 6 every second [13].

SPSM'15, October 12, 2015, Denver, Colorado, USA.
 ISBN 978-1-4503-3819-6/15/10 ... $15.00.

Malware developers are looking for ways to distribute their malwares. In the world of mobiles, one of the common approaches is repackaging [22]. In repackaging, the malware developer usually downloads a popular app, repackages it, recompiles it if needed, and embeds the malicious code into the app's code. The new code will be zipped again, signed and published in an app store. In fact, attackers use the popular apps in order to increase the possibility of spreading malicious apps. A recent study has shown that more than 85% of malware samples were a repackaged version of legitimate apps [15]. Repackaging provides a way to change the code for variety of purposes, not just for embedding malwares. It has become a serious concern for app developers since they may lose revenues in case their paid app is repackaged and published in third party markets. Some free apps also gain revenue by advertising in part of their app. The repackaged app can be manipulated to change the advertisements. Monetary loss is not the only effect. Repackaging may even ruin the company's reputation as shown in [6].

As repackaging is a popular way to broadcast malware, in this paper, we investigate how malicious code can be embedded into legitimate apps. More precisely, we focus on the services and notifications that malicious apps provide as opposed to legitimate apps. The use of services stems from the fact that malware needs to continue operating even if the original app terminates. In addition, malicious apps need to provide users with the same experience as when using normal apps. Therefore, any malicious operation needs to be designed in a way that goes undetected by users.

To achieve this, we examine carefully the life cycle of services in Android apps. Based on this, we extract a number of features that characterize the services of apps. We use these features to classify over 250 legitimate and malicious apps. The results show that:

- 68% of malwares that contain services require permissions and do not communicate with rest of the components in the apps after being started.
- 92% of the studied legitimate apps notify the user of the app about their operation of background services by sending notifications or by passing messages or by activating visible components in the app.
- Services in nearly all the malwares and more than 80% of the normal apps use permissions requested by the app.
- The apps that upload malicious code through updating do not have services.
- All the services in malwares are started and not bound by other components of the apps.

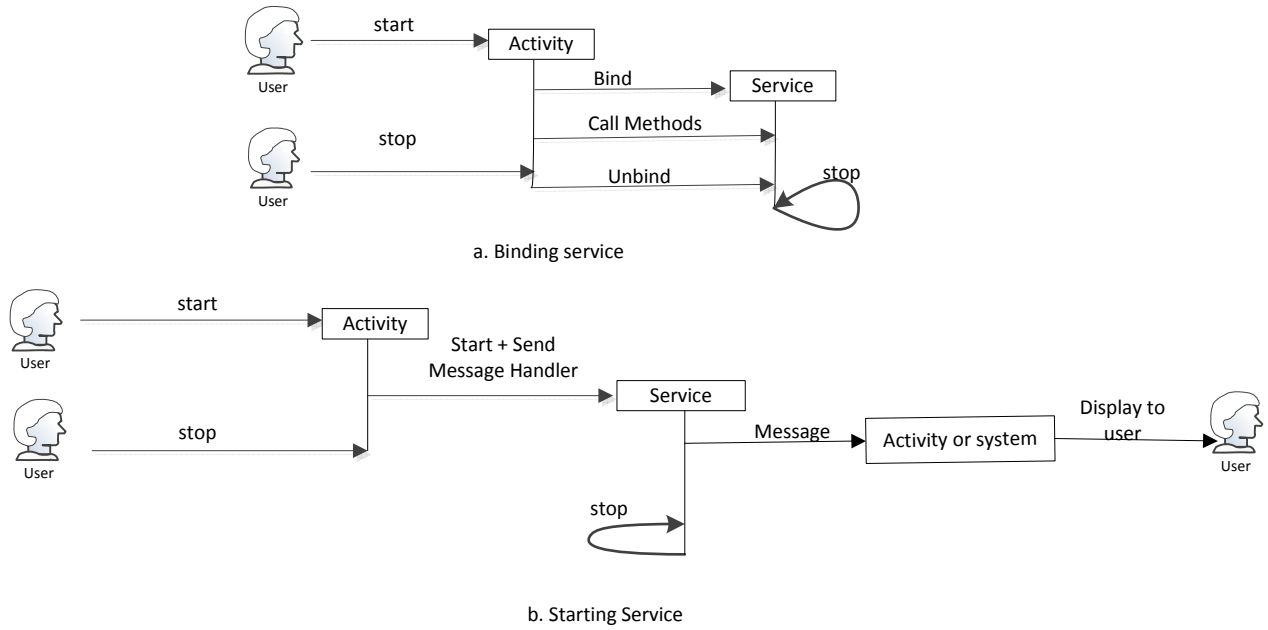

Figure 1. Android apps' service life cycle

2. LIFE CYCLE OF APP'S SERVICES

A typical Android app contains four types of components: Activities, Services, Content providers and Broadcast receivers. Activities represent what the user can do with the app. A service is a component that runs in the background. A content provider manages a shared set of app's data. A broadcast receiver is a component which responds to system-wide broadcast announcements such as "the battery is low".

Activities, services and broadcast receivers are activated by an asynchronous message object called intent. More details about components are available in Android developers' official site [4].

From the perspective of the user who runs an app, the app's components can be divided into two parts. The first part consists of the foreground part, which contains activities that a user can use. Activities are, in fact, components that should provide the same experience to the user in both between the repackaged and the original app. The second part of the app is the background components including services, broadcast receivers and content providers operating in the background and the operation's result is send to the user in activities. By this categorization, if a malware wants to operate for a long time, hidden from the user, it should be added to background components. This is why, in this study, we focus on analyzing the life cycle of services.

In AndroidManifest.xml, a service is defined by a <service> tag. It has attributes such as name, *exported* attributes that show if other applications can use this service, and an intent filter, which lets the Android system respond to intents. The service is implemented in Java as the subclass of Service. The class name has to be similar to the name defined in AndroidManifest.xml for the service. A service in Android can started in two forms, *started* or *bound*. In the started form, a service is started by an another component of the app by calling method `startService()`. It will continue to run indefinitely in the background even if the user switches to other applications. It will be stopped by calling `stopSelf()` or `stopService()` methods. In the bound form, a component is bound to a service that it uses by calling the method `bindService()`. A bound service offers client-server interface, allowing the components to interact with each other. A bound service is destroyed as soon as another application component is unbound. Usually, for long running operations and single tasks, the `startService()` will be used.

Based on the nature of usage for two kinds of the service forms, started and bound, if a stealthy operation wants to be added to an app it can be embedded in service and start it with `startService()`. Thus, it will have a chance to run indefinitely in a device. Since it is the user who decides which activity should operate while running an app, the bound service, whose public methods are called by the activity, is indirectly controlled by the user. But when an activity runs a service, the service can continue running even if the user quits running the app.

When the implemented task for a service is finished, the service usually notifies the user of its completion. Notifying a user of a service operation can be performed using these approaches:

- By generating a notification object and passing it to the system. Notifications may contain actions, which can start an activity.
- By starting an activity since activities are the foregrounded components and users can see the results of the task done in services.
- Through message passing by having the component give permission to the service to send a message by passing message-handler to the service while starting it. Some protocols such as AIDL are designed to let other applications use the service of other applications. They are also based on the message handling approach.

The Android system tries to keep an application process alive as long as possible, but sometimes it needs to kill some processes to reclaim memory for new processes. It terminates processes based on their level of importance to the user. For example, the

processes containing activities the user interacts with are given highest priority. It should be noted that services that are terminated may be restarted as standalone with a starting form, instead of a bound form.

Considering the event-driven nature of Android applications, we can group the operations performed in a service into two categories. The first one contains all operations that can start after the service is created by an activity. It will continue to operate, and finish after finishing its operation. The user can be notified of the finished task. The second category contains tasks that are executed in special situations. For example, the service start to work when a broad cast receiver start them when a particular even happen such as the event of receiving messages or emails. In fact, it is the system who indirectly started the service.

A broadcast receiver is defined through the <receiver> tag in AndroidManifest.xml of the application. It also contains an attribute, set to “true” if it can receive messages from sources outside its application, “false” otherwise. The default value depends on whether the broadcast receiver contains intent filters. The absence of any filters means that it can be invoked only by Intent objects that specify its exact class name. So the default value is "false". On the other hand, the presence of at least one filter implies that the broadcast receiver is intended to receive intents broadcasted by the system or other applications, so the default value is “true".

When a receiver is defined by intent filter, it lets the other app or components of the app send implicit intents to this receiver. However, an app can define the intent filter in the receiver to obtain the implicit intents that may not target the particular application, for example, getting alarms or receiving messages from the system. It can also be a way to restart a service or activity inside an app. As we will show in the next section, we will examine this feature to see if malicious apps behave in such a way or not. Note that a broadcast receiver can only be started; it cannot be bound to a service.

Figure 1 illustrates the communication of a service during its life cycle. This figure shows how a user as an actor in the lifecycle of components can start a service and be notified from the service operation. It should be noted that in this figure, instead of activity, there can be any other type of app’s component.

As shown in Figure 1, the user’s role is important for this study since malware that attempts to hide itself, would operate stealthy by avoiding any communication with the user. A suspicious behavior may motivate users to use expert tools to detect and remove malwares. In addition, malwares that perform malicious operations that involve user interactions such as sending messages to user contacts or alarm the user of erasing the files are usually not published on reliable app stores such Google play store [18]. They are published on third party (untrusted) app stores.

With the focus on services of an app, we can define each app by the set of services it has. Each service consists of some features which characterize the app’s behavior. The defined features are summarized as follows:

- Service permissions used by the service; a service that does not need any permission does not perform any malicious operation.
- Notifications generated by a service to show if it has interactions with the user; it can be done by generating notifications or sending messages to any message handler, which is passed to a service while starting it.
- List of activities which start the service or bound to the service; if a service is bound to a component, it has access to its public method. Thus, it is indirectly connected to other components and will die as soon as other components unbind it.
- List of activities started directly by the service; as activities are visible to users, it can be a way of notifying the user of the operation of service.
- List of broadcast receivers in the app that start a service; these broadcast receivers are defined by their intent filters. They can help determine when a service starts to perform actions. Note that a broadcast receiver cannot be bound to a service.

The list of services is easily extracted from AndroidManifest.xml file. The tag <uses-permissions> in AndroidManfest.xml shows that permissions are needed by the app. These tags are usually defined independently from the components, making difficult to know which components use which permissions. Thus, we need to study the application code and APIs used in each class to identify the permissions used by a service. In the AndroidManifest.xml, each component android:name attribute specifies the class name of the component in the corresponding Java code.

We used a tool called Androguard[1] to extract the methods called in a class and permissions used by them. We wrote a python script over Androguard that recursively follows the methods called in a service class, starting from callback method onStartCommand() and onBind(). All the permissions used in these methods specify the permissions of the service.

We were also curious to know if a service notifies the user of its operations. As such, we extracted all the notification objects generated in the methods called during the service life cycle. Apart from that, we also extracted direct call for activities. As activities run in the foreground, they inform the user of the operations done in service. As discussed in the previous section, our assertion is that stealthy services don’t do this. In this regard, we also extract the messages passed by the service to other components. The tool Androguard let us find the object Notification and Message in a service. The other feature needed to be extracted was the broadcast receivers and activities, which call the service as well as activities called by the service. When a service is started by a broadcast receiver, it shows the special situation in the device that a service is called, such as “a message received” or “battery is low”.

3. EXPERIMENT

3.1 Dataset

To investigate our assertion, we studied a set of malwares to observe how they will behave. We also studied a set of normal applications to see if their services behaved differently. The Genome malware dataset [5] was used for our study. The legitimate apps are downloaded from Google play store in April 2015. We downloaded 200 apps randomly from the first twenty categories in Goggle play store; ten apps from each category.

[1]https://code.google.com/p/androguard/

3.2 Analysis and results

As we explained, malwares try to hide themselves by running in background services and having no communication. We can classify the feature of the services as follows. The first feature, PERMISSION, represents the permissions used during the service life span. The second, MESSAGE, is the notifications or messages passed to activities or shown to the user. These two features were extracted just by running a script that we have developed.

We also needed to extract two other features. The first one, CALL_ACTIVITY, shows if the service calls other activities after finishing its operation informing the user when the operation is done. The second one, SERVICE_BOUND, depicts if the service is bound because the bound services are alive until the component, which binds them is alive. To extract CALL_ACTIVITY and SERVICE_BOUND and study the apps' "smali" code[2], assembly for the dex format, we used APKTOOL[3].

We extracted the features for all legitimate apps downloaded from Google play store and the malware dataset. The results are shown in Table 2. We use the following labels to refer to groups of the various features of services:

- A: NO PERMISSION
- B: PERMISSION and MESSAGE
- C: PERMISSION and NO MESSAGE and NO CALL_ACTIVITY and NO SERVICE_BOUND
- D: PERMISSION and NO MESSAGE and (CALL_ACTIVITY or SERVICE_BOUND)
- E: NO SERVICE
- F: NO IMPLIMENTATION

Note that in this table, having "NO" before the feature's name means that the app contains at least one service that does not have this feature. For example NO MESSAGE means that the app contains a service which does not send messages or notifications.

While studying the apps' services, we have found that some of the apps do not have any services and as such, we categorized them as NO SERVICE. There were some apps that contain only definition for a service in AndroidManifest.xml but the service does not implemented in the source code they have been categorized as NO IMPLEMENTATION. This may be due to the developer's mistake in keeping unnecessary service definitions in AndroidManifest.XML.

It should be noted that if there is an app containing services with different features, we take into account the most restrictive services. For example, if a normal app contains two services where the first one is PERMISSION and MESSAGE and the second one is PERMISSION and NO MESSAGE and CALL_ACTIVITY, we put it in the third group in Table 1.

Apps that contain services with NO PERMISSION show that they have safe services. Clearly, a service performing malicious operation needs permissions. As it is shown in Table 1, the number of these apps in the legitimate apps dataset is 25 out of 200 (12%) and in the malicious app dataset is 1 out of 65 (1.5%).

The number of apps with PERMISSION, NO MESSAGE, CALL_ACTIVITY shows a significant difference between the apps in the malware dataset and those in the legitimate app dataset.

The number of apps in Group D which shows the services who are bound to a component and will stop after the component unbind it; and in group B which shows the services with MESSAGE are significantly large in normal apps. On the other hand, the number of apps in group C, which shows the apps with services that have no connection with rest of the app after being started, are larger than other groups in the malware dataset. These results show that malwares and normal apps have different behavior with respect to the service lifecycle and its connections and communication with the rest of the app.

In order to have a better understanding of malicious apps that have NO SERVICE or NO PERMISSION, we further examined sample malicious apps. We use the code of malware and information provided by Zhou et al. [22] and by Felt et al. in [9].

Among the group of malwares that do not have services, FakeNetFlix and FakePlayer ask for user credential information directly from the user via an activity component. This sort of malwares is easily detectable by a security expert. DroidDeluxe and some version of Asroot do not have malicious operations. They get root privilege by exploiting a vulnerability during installation. Similarly, DroidKungFuUpdate, AnServerBot, BaseBridge and Plankton get root privilege to download and install malicious apps. SMSReplicator, Walkinwat, YZHC do not have services and do the malicious operation when an event is received by a broadcast receiver. For example, SMSReplicator has a broadcast receiver that listens to the incoming message and forward it to the selected number.

AnserverBot is a malware that asks users for update and installs the malicious payload. Therefore, the services in the malware itself do not need permissions.

ADAR, which has PERMISSION and MESSAGE, use media player to send notifications. One version of Asroot and BaseBridge also use notifications while updating for malicious payload. SNDApps notification is sent by a service in the repackaged app that the malicious payload added to the original app. It was the only sample in the malware dataset that adds the malicious operation to the existing service of a legitimate app. However, there was another service related to malware in particular. This shows that studying the content of messages in a service can help detect suspicious services.

4. RELATED WORK

The increased rate of Android apps has been accompanied by an increase in malware spread in app stores. Below are some security issues raised by the increase of malware:

- Users are not aware that by giving some permission to an app, they may cause security issues.
- Attackers have the same capability to develop malware and upload them in marketplaces as legitimate developers.
- Although official Android app store sites investigate apps before uploading them to the store it is not clear how these investigations are carried out and to what extent. In [6], the authors show that some malware were uploaded in Android market place such as DroidDream Trojan in 2011.

[2]https://code.google.com/p/smali/
[3]http://ibotpeaches.github.io/Apktool/

Table 1. Results of classifying Apps based on their services

Category	**A:** NO PERMISSION	**B:** PERMISSION and MESSAGE	**C:** PERMISSION and NO MESSAGE and NO CALL_ACTIVITY and NO SERVICE_BOUND	**D:** PERMISSION and NO MESSAGE and (CALL_ACTIVITY or SERVICE_BOUND	**E:** NO SERVICE	**F:** NO IMPLIMENTATION
Normal (200)	25 (12%)	58 (29%)	14(7%)	76 (38%)	19 (9.5%)	8 (4%)
Malwares (65)	1 (1.5%)	3 (5%)	39(60%)	14(21%)	8 (12%)	0 (0%)

The increased number of malware in Android apps has motivated researchers to work on developing several detection techniques. Studies conducted in detecting malwares in desktop applications are used to study the Android apps in order to identify malicious behavior of the apps. Studying the similarity of apps is a detection approach as malwares in Android use repackaging to embed the malware in legitimate code [22]. There is no perfect method that could provide the desired accuracy in detecting malware and the most common issues are summarized as follows.

Struggles related to the use of machine learning algorithms: Some approaches [1, 3, 10, 14, 19, 20] use machine learning algorithms to learn the characteristics and behavior of malware and build clusters of malware families to detect zero-day malwares. Features used to learn malware behaviors are permissions [7, 19], intent [20], API [3, 19], system calls and smartphone features such as battery usage, memory, CPU and Network [2]. The main drawback of these approaches is that they need to have more than one malware sample in a family to learn their behavior.

Dynamic loading and Native code: Some malware like Base-Bridge and DroidKungFu Android malware [23] extract the actual malicious payload from external places rather than the original applications themselves. Thus, static analysis approaches [19] cannot detect them. To detect the malicious operations in native codes, it is suggested to study the OS interactions [18]. Detecting malicious code in dynamic loading and native code in Android apps still suffer from computational complexity [24].

Obfuscation: Detecting malware based on signatures has always suffered from the problem of obfuscation. To this end, behavioral graph models are used to learn malware behavior [8, 11, 12, 17, 20, and 23]. In these studies, the graph is used to find the structure of malware behavior, which is different from normal apps. Several studies have been proposed [8, 11, and 23]]. These approaches suffer from computation and memory usage overhead. Moreover, they fail to identify certain usages of instances/methods, which are encrypted or use Java reflection and native code.

Curse of studying the similarity of Apps:. Some studies focus on detecting the similarity of apps by analyzing the user interface of the apps. Since the two apps (the original and the repackaged one) are designed to offer the same experience and have a high chance of being downloaded the repackaged version has the same UI as the original version of the app. Zhang et al. [21] extracted the graph containing the activity component of the app and compared the similarity of the graph, extracted from the original and the repackaged apps. Shao et al. [15] followed the same idea but instead of using the activity component, they studied the similarity of resources in the two apps. Zhou et al. [24] considered that repackaged apps are published in third-party app stores and studied the instruction sequences in apps and measured the similarity of apps based on similar instructions. They found that the similarity between apps in the official Android marketplace and in third-party market places. Besides the computational overhead of these approaches, the main constraint is that they need to compare apps two by two to find the similar ones. They also cannot detect if the repackaged version contains malware or just minor changes such as ads, or contain no changes and are just resigned and published in third party app stores.

5. CONCLUSION AND FUTURE WORK

In this paper, we examined the service life cycle of apps to understand how malicious apps due to repacking and normal apps vary in terms of the services they offer. We found that malicious apps tend to start a service to perform malicious operations and have no connection to the other components of the app. However, services in normal applications are bound to other components and send message and notifications to users.

We intend in the future to continue examining the variations that exist between malicious apps and normal apps by studying their services. The results should lead to effective techniques for detecting malicious apps that would not require comparing apps in a large store of apps. We are always investigating other features besides service features.

6. ACKNOWLEDGMENT

This research is partly supported by a grant from Natural Sciences and Engineering Research Council of Canada (NSERC), Defence Research and Development Canada (DRDC) Valcartier (QC), and Ericsson Canada.

7. REFERENCES

[1] 2011 mobile threats reports, Juniper Networks, 2012.

[2] Alam, M. S., & Vuong, S. T. 2013. Random Forest Classification for Detecting Android Malware. In *Green Computing and Communications (GreenCom), 2013 IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing* (August 2013), 663-669.

[3] Alazab, M., Venkataraman, S., and Watters, P. 2010. Towards understanding malware behaviour by the extraction of API calls. In Cybercrime and Trustworthy Computing Workshop (CTC), 2010 Second, (July 2010), IEEE, 52-59

[4] Android Developer Guide, 2015, http://developer.android.com/guide/components/fundamentals.html

[5] Android Malware Genome Project, 2015, www.malgenomeproject.org.

[6] Arxan Technologies Inc. State of security in the app economy. http://www.arxan.com/resources/state-of-security-in-the-app-economy/

[7] Barrera, D., Kayacik, H. G., van Oorschot, P. C., and Somayaji, A. 2010. A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 73-84.

[8] Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. 2011. Crowdroid: behavior-based malware detection system for android. In Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (October 2011), ACM, 15-26.

[9] Felt, A. P., Finifter, M., Chin, E., Hanna, S., and Wagner, D. 2011. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ACM, 3-14.

[10] Islam, R. and Altas, I. 2012. A comparative study of malware family classification. In Information and Communications Security, 488-496. Springer Berlin Heidelberg.

[11] Gascon, H., Yamaguchi, F., Arp, D., and Rieck, K. 2013. Structural detection of android malware using embedded call graphs. In Proceedings of the 2013 ACM workshop on Artificial intelligence and security, (November 2013), 45-54. ACM.

[12] Luoshi, Z., Yan, N., Xiao, W., Zhaoguo, W., and Yibo, X. 2013. A3: Automatic Analysis of Android Malware. In 1st International Workshop on Cloud Computing and Information Security. Atlantis Press. (November 2013).

[13] McAfee Labs Reports, February 2015, http://www.mcafee.com/ca/resources/reports/rp-quarterly-threat-q4-2014.pdf

[14] Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., and Bringas, P. G. 2012. On the automatic categorization of android applications. In Consumer Communications and Networking Conference (CCNC), (January 2012). IEEE, 149-153.

[15] Shao, Y., Luo, X., Qian, C., Zhu, P., and Zhang, L. 2014. Towards a scalable resource-driven approach for detecting repackaged Android applications. In *Proceedings of the 30th Annual Computer Security Applications Conference* (December 2014), ACM, 56-65.

[16] Statistics and Market Data on Mobile Internet & Apps, May 2015,http://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

[17] Suarez-Tangil, G., Tapiador, J. E., Peris-Lopez, P., and Blasco, J. 2014. Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Systems with Applications*, *41*(4), (2014), 1104-1117.

[18] Tam, K., Khan, S. J., Fattori, A., and Cavallaro, L. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors. In *Proc. of the Symposium on Network and Distributed System Security (NDSS)*.

[19] Wu, D. J., Mao, C. H., Wei, T. E., Lee, H. M., and Wu, K. P. 2012. Droidmat: Android malware detection through manifest and API calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, (2012, August), 62-69.

[20] Yang, C., Xu, Z., Gu, G., Yegneswaran, V., and Porras, P. 2014. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *Computer Security-ESORICS 2014*, 163-182.

[21] Zhang, F., Huang, H., Zhu, S., Wu, D., and Liu, P. 2014. ViewDroid: towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks* (July 2014), ACM, 25-36.

[22] Zhou, Y. and Jiang, X. 2012. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (May. 2012), 95-109.

[23] Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. 2012. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In NDSS, Internet Society, (February 2012).

[24] Zhou, W., Zhou, Y., Jiang, X., & Ning, P. 2012. Detecting repackaged smartphone applications in third-party android marketplaces. In Proceedings of the second ACM conference on Data and Application Security and Privacy (February 2012), 317-326. ACM.