ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 3, Issue 2, pp: (654-661), Month: April - June 2015, Available at: www.researchpublish.com

Study of Issues, Malware Penetration and Defenses with Android Security Extension

[1]Mallikarjunaiah K M, [2]Mahesh .K Kaluti

[1]PG Student, [2]Asst. Professor, [1,2]Dept. Of CSE, AIET

***Abstract*: Three hundred and fifty thousand Android phones are activated each day. It is easy for third-parties to develop and distribute applications, since by the open source from Google. This will the same applies to malicious applications that pose a real threat to users' privacy. Increased popularity of the Android devices and associated monetary benefits attracted the malware developers, resulting in big rise of the Android malware apps. In this survey, we discuss the Android security enforcement mechanisms, threats to the existing security enforcements and related issues, malware growth timeline. This study introducing the enhancement of current security policy by adopting different components. This methodology provides a platform to the researchers for the next generation Android security.**

***Keywords:* Android Malware, Static Analysis, Dynamic Analysis, Malware.**

I. INTRODUCTION

Android shook the world of smartphones. It created an ecosystem full of developers eager to embrace and make use of the opportunities offered by smartphones. Thanks to the open model supported by Google, developers are not forced to pay high code certification fees or sharing a significant percentage of their profit with application distribution points (a.k.a. "markets"). The fast growth of Android smartphone operating systems's market shares is the tangible proof a real demand was met [4]. Not surprisingly, also the number of applications developed for Android is knowing a similar growth rate [3].

Unleashing the smartphone application market was beneficial also for end-users who can now choose among a huge variety of applications. While most of these applications achieve their goals without abusing users' privacy, some of them have recently hit the media for being quite effective in doing the opposite [5], [1]. Furthermore, an open and popular platform as Android, provides a perfect environment to exploit and disseminate security attacks.

Android security model mostly relies on application sandboxing. However, it has been reported that this model is vulnerable to privilege spreading attacks [13]. In this type of attacks, an unprivileged application exploits the permissions of privileged applications. If the malicious application is leveraging a vulnerability of a legitimate application then this type of attacks is often referred to as confused deputy attacks [12]. However, given that developing applications for the Android Market is quite simple (just pay a $25 fee), designing colluding applications that on purpose provide to other applications permissions without the user being aware of it is becoming increasingly popular.

Android popularity has encouraged the developers to pro-vide innovative applications popularly called apps. Google Play, the official Android app market, hosts the third party developer apps for a nominal fee. Google Play hosts more than a million apps with a large number of downloads each day [5]. Unlike the Apple appstore, Google Play does not verify the uploaded apps manually. Instead, official market depends on Bouncer [6], [7], a dynamic emulated environment to control and protect the market place from the malicious app threats. Though Bouncer protects against the malware threats, it does not analyze the vulnerabilities among uploaded apps [8]. Malware authors take advantage of such vulnerable apps and divulge the private user information to inadvertently harms the app-store and the developer reputation. Moreover, Android open source philosophy permits the installation of third-party market apps, stirring up dozens of regional and

ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 3, Issue 2, pp: (654-661), Month: April - June 2015, Available at: www.researchpublish.com

international app-stores [9]–[13]. However, the adequate protection methods and app quality at third-party app-stores is a concern.

Exponentially increasing malicious apps has forced the anti-malware industry to carve out robust and efficient methods suited for on device detection within the existing constraints. The existing commercial anti-malware solutions employ signature based detection due to its implementation efficiency [26] and simplicity. Signature based methods can be easily circumvented using code obfuscation necessitating a new signature for each malware variant [27], forcing the anti-malware client to regularly update its signature database. Due to the limited processing capability and constrained battery avail-ability, cloud-based solutions for analysis and detection have come into existence [28], [29]. Manual analysis and malware signature extraction requires sufficient time and expertise. It can also generate false negatives (FN) while generating signatures for the variants of known families. Due to the exponential increased malware variants, there is a need to employ automatic signature generation methods that incur low false alarms.

With the help of a program analyses, our research discovered 6 such Pileup flaws within Android Package Manager Service and further confirmed their presence in all AOSP (Android Open Source Project) [1] versions and all 3,522 source code versions customized by Samsung, LG and HTC across the world that we inspected. The consequences of the attacks are dire, depending on the *exploit opportunities* on different Android devices, that is, the natures of the new resources on the target version of an update. As examples, on various versions of Android, an upgrade allows the unprivileged malware to get the permissions for accessing voicemails, user credentials, call logs, notifications of other apps, sending SMS, starting any activity regardless of permission protection or export state, etc.; the malware can also gain complete control of new signature and system permissions, lowering their protection levels to "normal" and arbitrarily changing their descriptions that the user needs to read when deciding on whether to grant them to an app; it can even replace the official Google Calendar app with a malicious one to get the phone user's events, drop Javascript code in the data directory to be used by the new Android browser so as to steal the user's sensitive data, or prevent her from installing critical system apps such as Google Play Services.

II. STRUCTURAL OVERVIEW

The Android software stack as shown in figure can be subdivided into five layers: The kernel and low level tools, native libraries, Android Runtime, the framework layer and on top of all the apps.

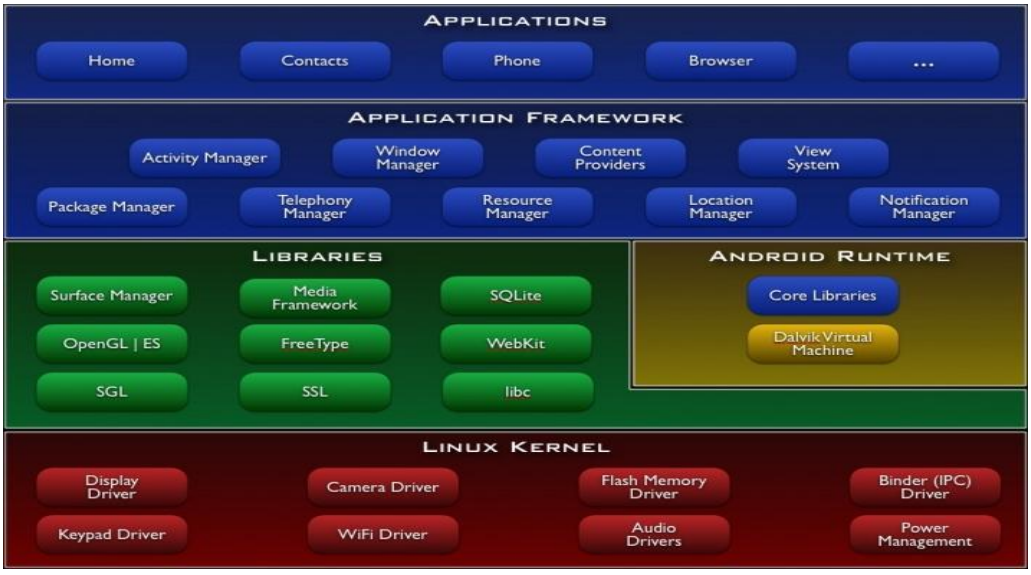

Fig 1: Android system architecture

The kernel in use is a Linux 2.6 series kernel, modified for special needs in power management, memory management and the runtime environment. Right above the kernel run some Linux typical daemons like bluez for Bluetooth support and wpa supplicant for WiFi encryption.

The Android Runtime consists of the Dalvik virtual machine and the Java core libraries. The Dalvik virtual machine is an

ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 3, Issue 2, pp: (654-661), Month: April - June 2015, Available at: www.researchpublish.com

interpreter for byte code that has been transformed from Java byte code to Dalvik byte code. Dalvik itself is compiled to native code whereas the the core libraries are written in Java, thus interpreted by Dalvik.

Frameworks in the Application Framework layer are written in Java and pro-vide abstractions of the underlying native libraries and Dalvik capabilities to applications. Android applications run in their own sandboxed Dalvik VM and can consist of multiple components: Activities, services, broadcast receivers and content providers. Components can interact with other components of the same or a different application via intents.

III. APPLICATIONS AND TASKS

Android applications are run by processes and their included threads. The two terms task and application are linked together tightly, given that a task can be seen as an application by the user. In fact tasks are a series of activities of possibly multiple applications. Tasks basically are a logical history of user actions, e.g. the user opens a mail application in which he opens a specific mail with a link included which is opened in a browser. In this scenario the task would include two applications (mail and browser) whereat there are also two Activity components of the mail application and one from the browser included in the task. An advantage of the task concept is the opportunity to allow the user to go back step by step like a pop operation on a stack.

***Application internals*:** The structure of an Android application is based on four different components, which are: Activity, Service, BroadcastReceiver and ContentProvider. An application does not necessarily consists of all four of these components, but to present a graphical user interface there has to be at least an Activity. Applications can start other applications or specific components of other applications by sending an Intent. These intents contain among other things the name of desired executed action. The IntentManager resolves incoming intents and starts the proper application or component. The reception of an Intent can be filtered by an application.

Intents, Intent filters and receivers Unlike ContentProviders, the other three component types of an application (activities, broadcast receivers and services) are activated through intents. An Intent is an asynchronously sent message object including the message that should be transported. Android utilizes different hooks in the application components to deliver the intents. For an activity, it's onNewIntent() method is called, at a service the onBind() method is called. Broadcast actions can be announced using Con-text.sendBroadcast() or similar methods. Android sends the Intent to the on Receive() method of all matching registered receivers.

APP STRUCTURE:

Android app is packaged into an APK .apk, a zip archive consisting several files and folders as shown illustrated in Figure 2. In particular, the AndroidManifest.xml stores the meta-data such as package name, permissions required, definitions of one or more components like Activities, Ser-vices, Broadcast Receivers or Content Providers, minimum and maximum version support, libraries to be linked etc.. Folder res stores icons, images, string/numeric/color constants, UI layouts, menus, animations compiled into the binary. Folder assets contain non-compiled resources. Executable file classes.dex stores the Dalvik bytecode to be executed on the Dalvik Virtual Machine. META-INF stores the signature of the app developer certificate to verify the third party developer identity.

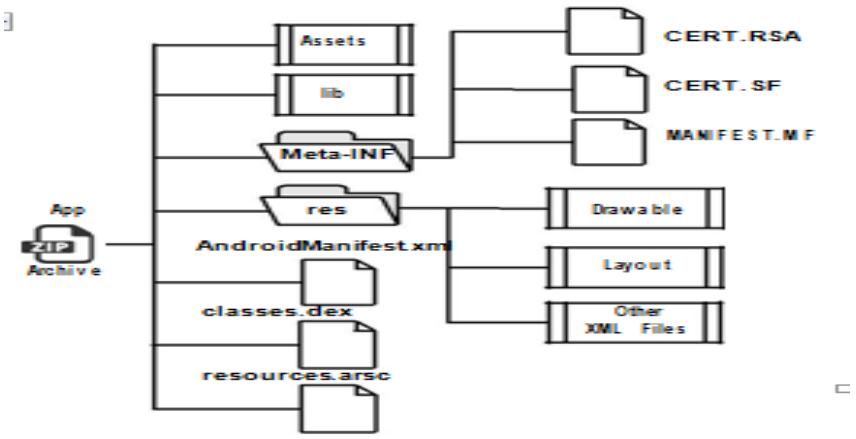

Fig. 2: Android PacKage (APK) Structure

ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 3, Issue 2, pp: (654-661), Month: April - June 2015, Available at: www.researchpublish.com

App Sandboxing: Android has been designed as secure mobile OS with a motive to protect the user data, developer apps, the device, and the network [34]. However, the security depends on the developer willingness and capabilities to adhere the best development practices. Also, user must be aware of the effect an app may have on the data and device security. For example, anti-malware apps have a restricted scanning and/or monitoring capabilities and/or file-system in the device. This section covers the Android security features.

Android Kernel implements the Linux Discretionary Access Control (DAC). Each app process is protected with an assigned a unique id (UID) within a isolated sandbox. The sandboxing restrains the other apps or their system services from interfering the other app. Android protects network access by implementing a feature Paranoid Network Security, a feature to control Wi-Fi, Bluetooth and Internet access within the groups [16]. If an app is permission for a network resource (e.g., Bluetooth), the app process is assigned to the corresponding network access id. Thus, apart from UID, a process may be assigned one or more group id (GIDs).

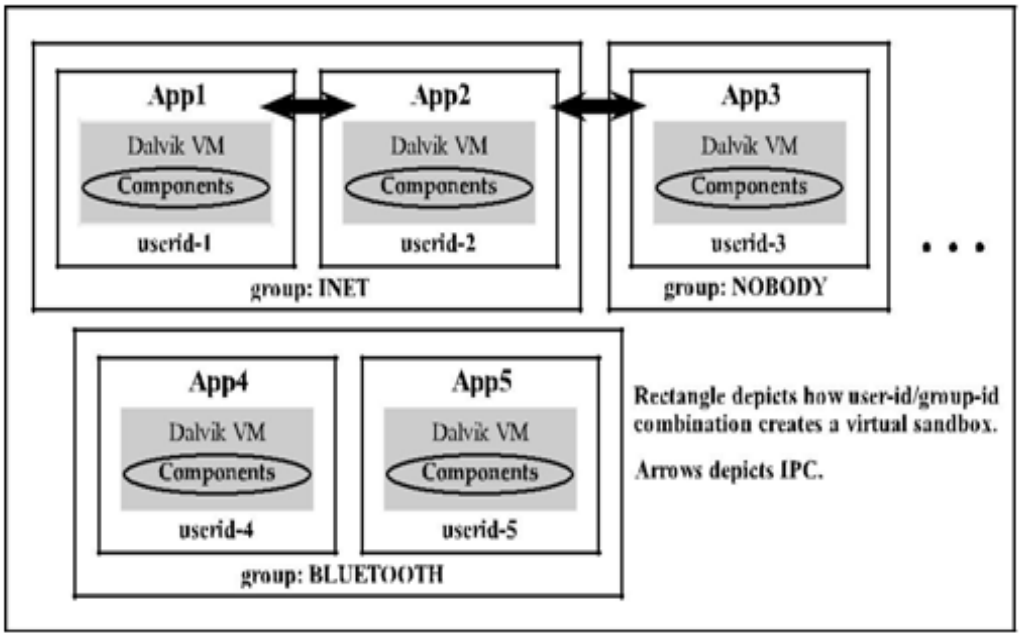

Fig. 3: Android Apps within Sandbox at Kernel-level

IV. ANDROID SECURITY ISSUES AND ENHANCEMENTS

AOSP is committed to a secure Android smartphone OS but, it is also susceptible to the social-engineering attacks. Once the app is installed, it may create undesirable consequences for the device security. Following is the list of malicious activities that have been reported or can be employed across subsequent Android versions. Privilege escalation attacks were leveraged by exploiting publicly available Android kernel vulnerabilities to gain root access of the device. Android exported compo-nents can be exploited to gain access to the dangerous permissions. Privacy leakage or personal-information theft occurs when users grant dangerous permissions to malicious apps and unknowingly allows access to sensitive data and ex-filtrate them without user knowledge and/or consent.

Malicious apps can also spy on the users by monitoring the voice calls, SMS/MMS, bank mTANs, recording audio/video without user knowledge or consent. Malicious apps can earn money by making calls or subscribe to premium rate number SMSes without the user knowledge or consent. Compromise the device to act as a Bot and remotely control it through a server by sending various commands to perform malicious activities. Aggressive ad campaigns may entice users to download potentially unwanted apps (PUA's), or malware apps . Colluding attack happens when a set of apps, signed with same certificate, gets installed on a device. These apps would share UID with each other, also any dangerous permission(s) requested by one app will be shared by the colluding malware. Collectively, these apps perform malicious activities, whereas, their individual function-ality is benign. For example, an app with READ_SMS permission can read SMSes and ask the colluding partner with INTERNET permission to ex-filtrate the sensitive information to a remote server.

ISSN 2348-1196 (print)
International Journal of Computer Science and Information Technology Research ISSN 2348-120X (online)
Vol. 3, Issue 2, pp: (654-661), Month: April - June 2015, Available at: www.researchpublish.com

V. MALWARE PENETRATION ON SURVIVAL TECHNIQUES

A. Repackaging Popular Apps:

Repackaging is a process of disassembling/decompiling the popular free/paid apps from the popular market places, inserts, appends the malware payload, re-assemble the trojan app and distribute them via the less monitored local app-stores. An app can be repackaged with the existing the reverse-engineering tools. Repackaging process is illustrated in Figure 4.

Repackaging is one of the most common malware app generation technique. More than 80% samples from the Malware Genome Dataset are repackaged malware variants [4] of the legitimate official market apps. Repacking and repackaging techniques can be used the generate large number of malware variants. It can also be used to generate a number of unseen variants of the already known malware. As the signature of each malware variant varies, the commercial anti-malware detect the unseen malware. Repackaging is a big threat as it can pollute the app distribution market places and also hurts the reputation of the third party developer.

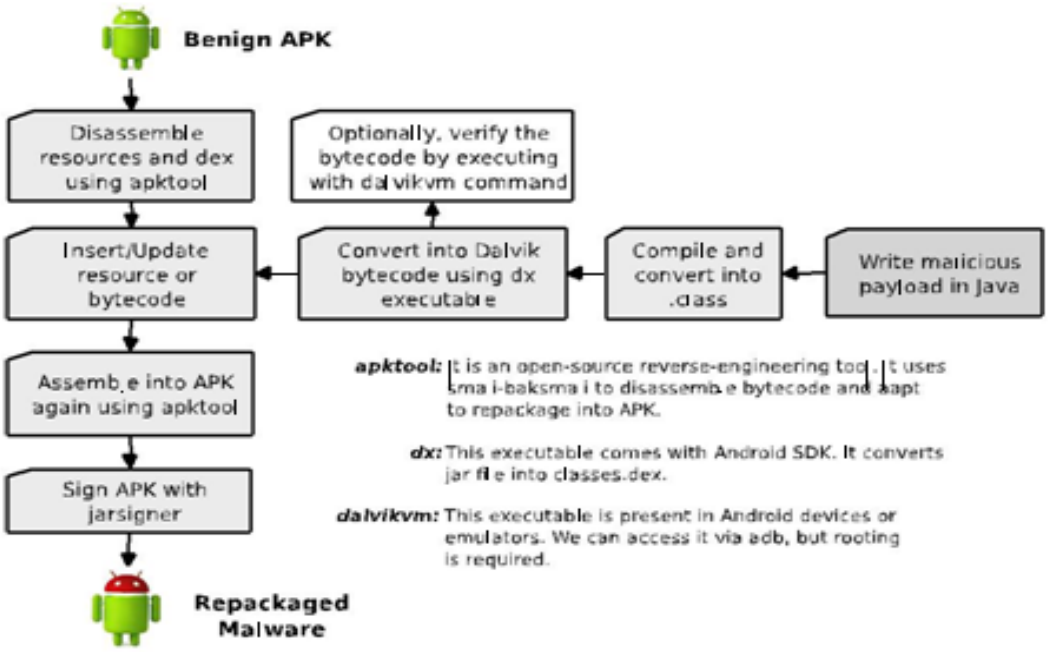

Fig 4: App Repackaging Process

In this scenario, the data shadowing approach of AppFence would have returned to OrgApp an empty list of contacts making OrgApp completely non functional for the user. If the user decides not to use data shadowing, then OrgApp would get access to the whole set of contacts, that is tagged with private and public label. When the OrgApp would send the contacts to the cloud service, AppFence would block the exfiltration of tainted data even if the OrgApp is sending the data to the url set by the user. Exfiltration blocking prevents tainted data to be written in a socket. Basically, when an output buffer contains tainted data AppFence drops the buffer *covertly*, misleading the application by indicating that the buffer has been sent, or *overtely*, by emulating the OS behaviour when the device is in airplane mode. Again, such a mechanism is too restrictive and would make the OrgApp useless to the user.

We do not expect that the average Android user is able to create policies when installing applications. For this rea-son, we have extended the Android installer with the User Setting Interface (USI) component (see Figure 1). When a new application is installed, the installer presents to the user the set of permissions that the application requires. These permissions are extracted from the application manifest file. The USI intercepts the permissions requested by an applica-tion and generates policies and labels according to type of permissions that the application is requesting. For instance, when the OrgApp is installed it requires access to the phone contacts (READ_CONTACTS) and internet (INTERNET). The first requirement will trigger the USI to generate a basic policy for OrgApp to access the contacts and the internet. The USI checks in the Labelling Store for possible extra labels associated with the data type requested from the application. For contacts two extra labels are specified for public and private entries.

B. Pileup Scanner:

With the flaws identified by the detector and the exploit opportunities collected by EOA, we can provide a service to Android users to detect the privilege escalation attempts aimed at the system upgrading process. Such a service is delivered through a Pileup scanner app that operates on the user's device and checks the attributes and properties of all her third-party apps to find out those suspicious. This approach can offer timely and less intrusive protection than patching all vulnerable PMSes, given the fact that this will affect possibly billions of Android devices, all manufacturers and carriers. Also importantly, it is less clear to us how to fix the problem through patching without causing any collateral damages to the user's live system: actually even months after we notified Google and Android about those vulnerabilities, they still have not deployed effective solutions to address all the problems we discovered.

C. Measurement of Opportunities:

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 EOA (Section IV-C) against a large number of Android images to understand the exploit opportunities (new exploitable attributes and properties) they bring in.

Landscape. 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 exploit 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 38 Google and 3,511 Samsung images we downloaded, we identified 741 update instances. The statistics on their total exploit opportunities in each instance are illustrated in Figure 5. Particularly, we found that 50% of those instances have more than 71 opportunities.

VI. FINE-GRAINED ACCESS CONTROL POLICIES

We start with some examples of policies for fine-grained control over applications accessing user data. Let us consider an Organiser application (OrgApp) providing a smart way of organising the user's contacts and a back-up capability that stores the user's contacts over a cloud service. For its functionalities, the OrgApp requires access rights to the phone contact provider and to the internet service. In this scenario, the user wants that only her work contacts are accessible to the OrgApp. Moreover, the user wants to make sure that the OrgApp sends the contacts to a specific location over the internet. The user's requirements can be expressed by two policies.

Preventing Access Right Spreading: One of the main security issues of the Android platform is the spreading of access permissions. Applications can implement services. These services can be invoked by other applications. In this way, the application implementing the service can allow other applications to use its permission to access phone resources. To make matter concrete, let us consider the following scenario. An application A requests permission to access the internet. The application A could pose as a simple application to provide news feeds and it would not look suspicious to the user. An application B that acts as a navigation application requires permission to access the GPS to display the user's current position. Moreover, application B implements a service that allows other applications to access the GPS through application B permission.

The results shown in Table I are averaged over 50 executions. The second row represents the execution time for the stock Android for performing a read of 100 contacts. The third row reports the time values for performing the same operations but this time with framework active. We have inserted a policy that filters out private data. However, since all contacts are set as public (0 private contacts) then no filtering is perform although the clause is executed. From the fourth row, we have increased the number of private contacts from 20 up to 80 (last row). We can see that the overhead introduced by quite high. However, these experiments have been executed on non-optimised data structures.

Native applications: Android applications that need more performance than the Dalvik VM can offer, can be partitioned. One part stays in the Dalvik VM to provide the application UI and some logic and the other part runs as native code. This way applications can take advantage of the device capabilities, even if Android or Dalvik do not offer a certain feature.

The native code parts of an application are shared libraries which are called through the Java Native Interface (JNI). The shared library has to be included in the applications .apk file and explicitly loaded. The code from the native library is loaded into the address space of the application's VM. This leads to a possible security hole, as the security means described in section 2.5 do not cover native code.

Other Promising Techniques: Third party app developers earn revenues on free apps by using the in-app advertisement libraries. A number of advertisement agencies provides the advertisement libraries to the app developers for inclusion in apps to earn revenues with targeted advertising. AdRisk detected a few aggressive ad libraries performing targeted advertisements at the cost of the user privacy. There have been instances of ad-affiliate networks getting classified as suspicious due to either targeted advertisement inclusions or sending malicious advertisement and compromise the user security [3]. Thus, it is equally important to detect such ad libraries within an app to make an informed decision. AdDetect is a promising semantic approach that detects the presence of in-app ad-library with reasonable accuracy compared to existing approaches.

Users also propose a machine learning model to predict the resource hoggers. Moreover, in the authors proposed a novel solution based on a behavior-triggering stochastic model to detect the target, and advanced malware.

SMS Trojans capable of sending messages to premium-rate numbers are growing to maximize monetary benefits. Elish et al. devised a static anomaly detection method to identify illegitimate data dependency between arguments of user input call-backs to sensitive functions. Using this approach they demonstrated the detection of some Android malware that send messages without user knowledge or consent. However, their approach does not take into account asynchronous APIs in Android such as inter-component communication, which fails to detect sophisticated SMS Trojans such as Dendroid [134]. AsDroid [111] is an another interesting static analysis tool that detects stealth behavior by finding semantic mismatch between the user-interface texts and their corresponding use of sensitive features.

VII. CONCLUSION AND FUTURE WORK

In this paper, we have presented Android security extension that supports fine-grained access control policies. Taint Droid taint mechanism for tagging data and enforcing security decisions on how data has to be disseminated within the device (application to application) or the outside world (through internet connections). Compared to other approaches, this is able to filter out data tagged with user-defined labels (such as public, private, confidential, etc.). In this way, applications can still access the data without reaching for user's sensitive information. Android is a core delivery platform providing ubiquitous services for connected smartphone paradigm, thus monetary gains have prompted malware authors to employ various attack vectors to target Android. Due to large increase in unique malware app signature(s) and limited capabilities within An-droid environment, signature based methods are not sufficient against unseen, cryptographic and transformed code. Researchers have proposed various behavioral approaches to guard the centralized app markets as malware authors are targeting easy-to-reach-user online distribution mechanism.

REFERENCES

[1] Android Security Study for the applications performances. http://www.appstore.org.

[2] Android Project. http://www.android.com.

[3] ARM Trustzone Technology. http://www.arm.com/products/processors/ technologies/trustzone.php.

[4] Gartner says android to command nearly half of worldwide smartphone operating system market by year-end 2012.

[5] AppBrain, Number of applications available on Google Play, http: //www.appbrain.com/stats/number-of-android-apps (Online; Last ac-cessed October 10 2014).

[6] Google bouncer : Protecting the google play market, http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-at-google-bouncer/ (Online; Last Accessed 15th October 2013).

[7] Android and security: Official mobile google blog, http://googlemobile.blogspot.in/2012/02/android-and-security.html (Online; Last Accessed 15th October 2013).

[8] E. Chin, A. P. Felt, K. Greenwood, D. Wagner, Analyzing Inter-Application Communication in Android, in: Proceedings of the 9th international conference on Mobile systems, applications, and ser-vices, MobiSys '11, ACM, New York, NY, USA, 2011, pp. 239–252. doi:10.1145/1999995.2000018. URL http://doi.acm.org/10.1145/1999995.2000018 Pandaapp, http://www.pandaapp.com/ (Online; Last Accessed 1st March 2014).

[9] Baidu, http://as.baidu.com/ (Online; Last Accessed 1st March 2014).

[10] Opera Mobile App Store, http://apps.opera.com/en in/ (Online; Last Accessed 1st March 2014).

[11] AppChina, http://www.appchina.com/ (Online; Last Accessed 1st March 2014).

[12] GetJar, http://www.getjar.mobi/ (Online; Last Accessed 1st March 2014).

[13] ESET - Trends for 2013, http://go.eset.com/us/resources/white-papers/Trends for 2013 preview.pdf (Online; Last Accessed 11th February).

[14] Kaspersky Security Bulletin 2013. Overall statistics for 2013, https://www.securelist.com/en/analysis/ 204792318/Kaspersky Security Bulletin 2013 Overall statistics for 2013 (Online; Last Accessed 11th February).

[15] McAfee Labs Threats Report: Third Quarter 2013, http://www.mcafee.com/uk/resources/reports/rp-quarterly-threat-q3-2013.pdf (Online; Last Accessed 11th February).

[16] F-Secure: Mobile Threat Report Q1 2013, http://www.f-secure.com/static/doc/labs global/Research/Mobile Threat Report Q1 2013.pdf (Online; Last Accessed 11th February).

[17] F-Secure: Mobile Threat Report Q3 2013, http://www.f-secure.com/static/doc/labs global/Research/Mobile Threat Report Q3 2013.pdf (Online; Last Accessed 11th February).

[18] F-Secure: Mobile Threat Report H1 2013, http://www.f-secure.com/static/doc/labs global/Research/Threat Report H1 2013.pdf (Online; Last Accessed 11th February).

[19] VirusTotal, https://www.virustotal.com/ (Online; Last Accessed 11th February 2014).