Android Security: A Survey of Issues, Malware Penetration, and Defenses

Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, *Senior Member, IEEE*, and Muttukrishnan Rajarajan

***Abstract*—Smartphones have become pervasive due to the availability of office applications, Internet, games, vehicle guidance using location-based services apart from conventional services such as voice calls, SMSes, and multimedia services. Android devices have gained huge market share due to the open architecture of Android and the popularity of its application programming interface (APIs) in the developer community. Increased popularity of the Android devices and associated monetary benefits attracted the malware developers, resulting in big rise of the Android malware apps between 2010 and 2014. Academic researchers and commercial antimalware companies have realized that the conventional signature-based and static analysis methods are vulnerable. In particular, the prevalent stealth techniques, such as encryption, code transformation, and environment-aware approaches, are capable of generating variants of known malware. This has led to the use of behavior-, anomaly-, and dynamic-analysis-based methods. Since a single approach may be ineffective against the advanced techniques, multiple complementary approaches can be used in tandem for effective malware detection. The existing reviews extensively cover the smartphone OS security. However, we believe that the security of Android, with particular focus on malware growth, study of antianalysis techniques, and existing detection methodologies, needs an extensive coverage. In this survey, we discuss the Android security enforcement mechanisms, threats to the existing security enforcements and related issues, malware growth timeline between 2010 and 2014, and stealth techniques employed by the malware authors, in addition to the existing detection methods. This review gives an insight into the strengths and shortcomings of the known research methodologies and provides a platform, to the researchers and practitioners, toward proposing the next-generation Android security, analysis, and malware detection techniques.**

***Index Terms*—Android malware, static analysis, dynamic analysis, behavioral analysis, obfuscation, stealth malware.**

Manuscript received May 14, 2014; revised October 16, 2014; accepted December 9, 2014. Date of publication December 30, 2014; date of current version May 19, 2015. This work was supported in part by the TENACE PRIN Project 20103P34XC funded by the Italian MIUR and in part by the Project "Tackling Mobile Malware with Innovative Machine Learning Techniques" funded by the University of Padua. The work of M. Conti was supported by Marie Curie Fellowship PCIG11-GA-2012-321980, funded by the European Commission for the PRISM CODE Project. The work of P. Faruki, M. S. Gaur, and V. Laxmi was supported in part by the Department of Information Technology, Government of India Project Grant "Security Analysis Framework for Android Platform".

P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor, and M. S. Gaur are with the Computer Engineering Department, Malaviya National Institute of Technology (MNIT), Jaipur 302 017, India (e-mail: 2012rcp9518@mnit.ac.in; vlaxmi@mnit.ac.in; gaurms@mnit.ac.in).

M. Conti is with the Department of Mathematics, University of Padua, 35122 Padua, Italy (e-mail: conti@math.unipd.it).

M. Rajarajan is with the School of Engineering and Mathematical Sciences, Electrical and Electronics Engineering, City University London, EC1V 0HB London, U.K. (e-mail: r.muttukrishnan@city.ac.uk).

Digital Object Identifier 10.1109/COMST.2014.2386139

I. Introduction

ANDROID smartphone OS has captured more than 75% of the total market-share, leaving its competitors iOS, Windows mobile OS and Blackberry far behind [1]. Even though smartphones were used in the previous decade, since 2008, iOS and Android smartphone OS has generated an enormous attraction among the users and developers worldwide. Smartphones have become ubiquitous due to a wide range of connectivity options such as GSM, CDMA, Wi-Fi, GPS, Bluetooth and NFC. Gartner smartphone sale report 2013 reports 42.3% in Android devices compared to the previous year [1]. The overall market share increased to 78% from 66%, substantial rise of 12% among the users. However, the nearest Android competitor iOS sale decreased 4%, from 19 to 15 percent. Ubiquitous Internet connectivity and availability of personal information such as contacts, messages, social network access, browsing history and banking credentials has attracted the attention of malware developers towards the mobile devices in general and Android in particular. Android malware such as premium-rate SMS Trojans, spyware, botnets, aggressive adware and privilege escalation attack exploits reported exponential rise apart from being distributed from the secure Google Playstore and well known third-party market places [2]–[4].

Android popularity has encouraged the developers to provide 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 harm 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 international app-stores [9]–[13]. However, the adequate protection methods and app quality at third-party app-stores is a concern [4].

Android security solution providers report an alarming rise of malware from just three malware families with 100 samples in 2010, to more than hundred families consisting 0.12–0.6 million unique samples in the quarter four, 2013 [14]–[19]. The number of malicious apps uploaded on VirusTotal [20] is increasing exponentially. Malware authors use stealth techniques,

dynamic execution, code obfuscation methods, repackaging and encryption [21], [22] to bypass the existing protection mechanisms provided by the Android platform and commercial anti-malware. Existing malware propagates by employing the above techniques and defeats the conventional signature-based approaches. The new techniques that adapt to the smartphone platform and provide timely response are an imminent need for the Android platform. Proactive methods to detect unknown malware employing in-frequent signature updates, in-contrast to one signature for each malware are desirable for Android.

Malware app developers gain smartphone control by exploiting platform vulnerabilities [23], stealing sensitive user information [21], to extract monetary benefits by exploiting the telephony services [24] or creating botnet [25]. Thus, it is important to understand their operational activities, working models and usage patterns to devise the proactive detection for mobile devices.

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 availability, 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.

Off-device malware analysis methods are needed to understand the malware functionality. Samples can be analyzed manually to extract the malware signatures. However, given the rapid rise of malware, there is an urgent need of the analysis methods requiring minimum human intervention. Automatic analysis helps the malware analyst generate timely response to detect the unseen malware. Static analysis can quickly and precisely identify malware patterns. However, it fails against code transformations, native code and Java reflections [30]. Thus, dynamic analysis approaches, though time consuming, is an alternative to extract malicious behavior of a stealth malware by executing them in a sandbox environment.

Academia and industry researchers have proposed solutions and frameworks to analyze, and detect the Android malware threats. Some of these are even available as open-source. These solutions can be characterized using the following three parameters:

1) *Goal* of the proposed solution can be either app-security assessment, analysis or malware detection. App-security assessment solutions determines the vulnerabilities, which if exploited by an adversary, harms the user and device security. Analysis solutions check for the malicious behavior within unknown apps, whereas detection solutions aim to prevent the on-device installation.
2) *Methodology* to achieve the above goals can be either *static* or *dynamic* analysis based approaches to detect malware. Control-flow and data-flow analysis are the examples of formal static analysis [29]. In *dynamic* analysis, apps are executed/emulated in a sandboxed environment, in order to monitor their activities and identify anomalous behaviors, that are otherwise difficult with static analysis.
3) *Deployment* of the above discussed solutions.

Existing smartphone security surveys review the state of the art considering the popular mobile OS platforms [31], [32]. However, this review paper focuses on Android platform, the most popular mobile device OS. La Polla *et al.* [32] surveyed the smartphone security threats and their solutions for the period 2004–2011, which has very limited coverage of Android.

Suarez-Tangil *et al.* [31] extended the work of La Polla *et al.* [32]. In particular, they concentrated on smartphone sensor feature based misuse attacks such as hardware, communication, sensors and system. Authors gave an insight into the misuse of specific Android features affecting the overall device security. Authors categorized the malware based on their attack goals, distribution, infection and privilege acquisition. On the contrary, this review categorizes the malware according to the commercial anti-malware industry terminology and provides an accurate description of malware infection rate and threat perception between 2010–2014.

In 2011, William Enck [33] studied the Android security mechanisms, particularly protection through permissions and security implications of inter-app communication. Moreover, author discussed other third-party Android platform hardening solutions, their benefits and limitations. In addition, the study also examined app security analysis proposals and presented future directions to enhance the Android platform security.

This paper aims to complement the former reviews by expanding the coverage of Android security issues, and malware growth between 2010–14. The paper discusses code transformation methods and strength and limitations of notable malware analysis and detection methodologies. In particular, this paper comprehensively cover stealth techniques used by malware authors to evade the detection by generating variants of the already known malware. Finally, we propose a hybrid Android malware analysis and detection framework, an insight into our future research directions. This survey paper is organized as follows.

Section II discusses the Android app architecture and security enforcement mechanisms employed to weaken the attack surfaces. Section III covers Android security issues in-spite of existing enforcements discussed in Section II. Section III covers major security enhancements in the subsequent Android versions to tackle the enumerated issues. Section IV presents the time-line illustrating notable Android malware families between 2010–2013 and categorizes them according to their functionality. Section V covers various penetration and stealth techniques employed by the advanced android malware.

Sections VI and VII categorize the prominent assessment, analysis, detection methods along with their deployment solutions respectively. Section VIII classifies the state of the art tools proposed by the academia and anti-malware industry according to their functionality covered in Section VI. This section discusses the strengths and drawbacks of well known analysis techniques, tools and summarize as per the functionality discussed in Table I. Comparison of the popular, web based analysis sandbox is illustrated in Table II. Finally, Section IX concludes this paper and proposes a hybrid malware analysis and detection framework as a recommendation for the future research directions.

II. Android App and Security Architecture

Android is being developed under Android Open Source Project (AOSP), maintained by Google and promoted by the Open Handset Alliance (OHA). It consists of the Original Equipment Manufacturers (OEMs), chip-makers, carriers and application developers. Android apps are written in Java, however the native code and shared libraries are developed in C/C++. Typical Android architecture is illustrated in Fig. 1. The bottom layer Linux kernel is customized specifically for the embedded environment consisting limited resources. Android is developed on top of Linux kernel due to its robust driver model, efficient memory and process management, networking support for the core services. Currently, Android supports two Instruction Set Architectures: 1) ARM, prevalent on smartphones, Tablets; 2) x86, prevalent among the Mobile Internet Devices (MIDs). On the top of the Linux kernel, the native libraries developed in C/C++ support high performance third-party reusable, shared libraries.

Android user app, written in Java language is translated to Dalvik byte code that runs under newly created runtime, the *Dalvik Virtual Machine* (DVM) as illustrated in Fig. 1. It is specifically optimized for the resource constrained mobile OS platform. Once the OS boot completes, a process known as *zygote* initializes the Dalvik VM by pre-loading the core libraries. Zygote then waits through a socket to load the newly forked processes. Zygote process speeds up the app loading the instances of libraries to be shared with the new loaded user apps. Finally, the application framework provides a uniform and concise view of the Java libraries to the app developer. Android protects the sensitive functionality such as telephony, GPS, network, power-management, radio and media as system services with the permission based model.

A. App Structure

Android app is packaged into an APK `.apk`, a zip archive consisting several files and folders as shown illustrated in Fig. 2. In particular, the `AndroidManifest.xml` stores the meta-data such as package name, permissions required, definitions of one or more components like Activities, Services, 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` contains non-compiled resources. Executable file `classes.dex` stores the

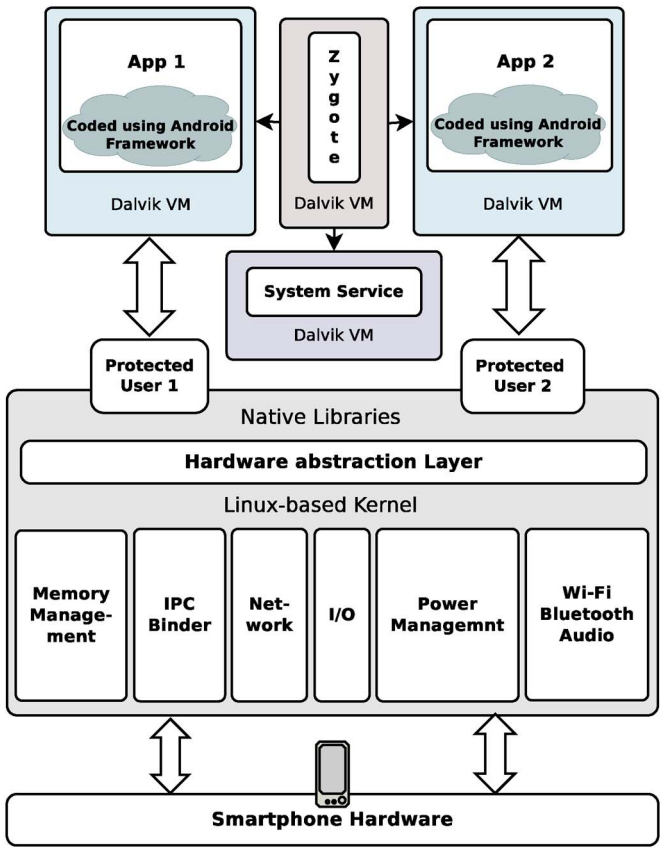


Fig. 1. Android Architecture [34].

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.

As mentioned previously, the Android apps are developed in Java. The development process is illustrated in Fig. 3. Compiled Java code generates a number of `.class` files, intermediate Java-bytecode of the classes defined in the source. Using the *dx* tool, `.class` files merged into a single *Dalvik Executable (.dex)*. The *.dex* file stores the Dalvik bytecode to be executed on the DVM to speedup the execution.

B. App Components

An Android app is composed of one or more components discussed below:

- *Activity*: It is the user interface component of an app. Any number of activities can be declared within the manifest depending on the developer requirements. Apart from some pre-defined task, an activity can also return the result to its caller. Activities are launched using the *Intents* as explained in the Section II-C.
- *Service*: Service component performs background tasks without any UI. For example, playing an audio or download data from the network. Services are launched using *Intents* further discussed in the Section II-C.
- *Broadcast Receiver*: This component listens to the Android system generated events. For example, `BOOT_COMPLETED`, `SMS_RECEIVED` etc. are system events. Other apps can broadcast their own application-defined events,

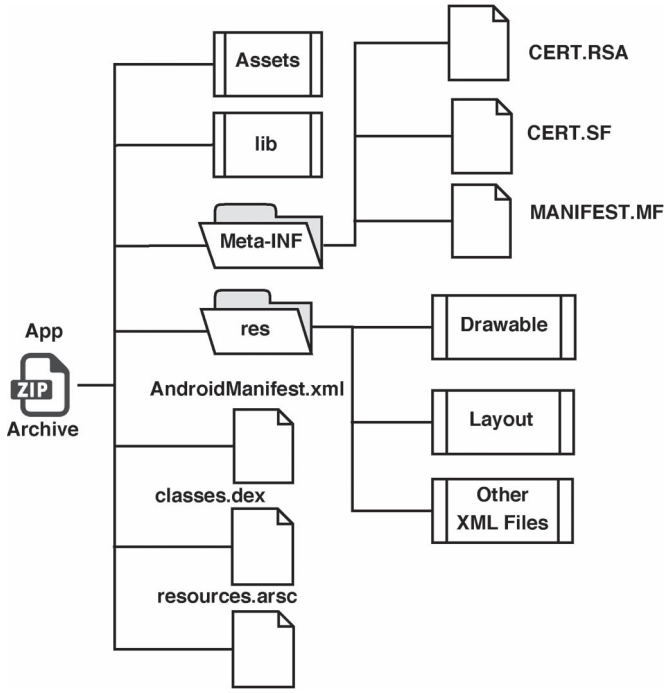


Fig. 2. Android PacKage (APK) Structure.

which can be handled by other the apps using the Service component.

- *Content Provider*: Content provider also known as the *data-store*, provides a consistent interface for data access between within and between different apps. Externally, the data within the content provider appears relational. However, it may have a completely different storage implementation. Data-store is accessible through the application-defined Uniform Resource Identifiers (URIs).

Component is made accessible to the other apps by explicitly exporting it. Listing 1 discusses the declaration of components as an usage example definition the `AndroidManifest.xml` binary. The declared component(s) can be invoked or executed independently since the app component development and communication is asynchronous. Android app has multiple entry-points, depending on the number of components an application defines.

C. Inter-Component Communication

Android Security protects apps and data with combination of system level and Inter Component Communication (ICC) [35]. ICC defines the core security utilizing the guarantee of the Linux framework. An app runs with a unique user-id to thwart the programming issues. Android middleware mediates the ICC between application and components. Access to a component is restricted by assigning an access permission label. When a component initiates ICC, the reference monitor looks at the permission labels assigned to its container app. If the target component access permission label is in the said collection, it allows ICC to be initiated. If the label does not belong to the collection, ICC establishment is refused even if the components are a part of same app. The developer assigns permission labels through the Manifest within an app. Developer defines the app security

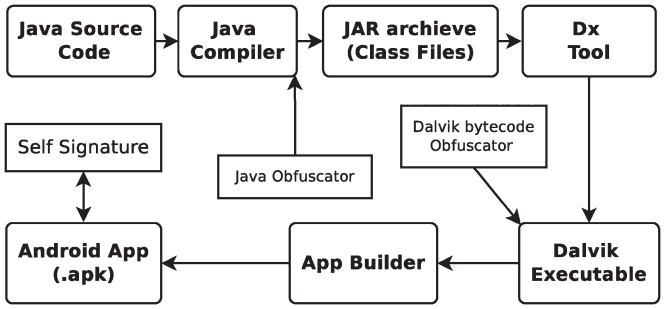


Fig. 3. App Building Process.

policy, whereas assigning permission to the components in an application specifies an access policy to protect its resources.

App components interact with each other at a high-level abstraction of inter-process communication (IPC) using *Intent*, handled by the Binder IPC driver. Apps invoke the *activities* and *services* and sends the broadcast events with Intents. System events are also broadcast through the Intents. Intent(s) can contain explicit address of the receiver components using class/package name field. Depending upon the presence of action, category and data fields, system sends implicit Intents to one or more matching receiver components. Each component registers itself to receive the Intent(s) using one or more *intent-filter*. It is also specified if the kind of action, category and/or data can be accepted by the intent. As shown in the Listing 1, *service* component is only invoked when it receives the system Intent with action equals to BOOT_COMPLETED in the Listing at line number 29.

D. 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. Anti-malware solutions do not have sufficient rights to perform aggressive malware checks due to enforced OS security model. 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 [36]. 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). Android app sandboxing is illustrated in Fig. 4.

An app must contain a PKI certificate signed with the developer key (see Fig. 3). App signature is the point of trust between Google and the third party developers to ensure the app integrity and the developer reputation. App signing procedure places an

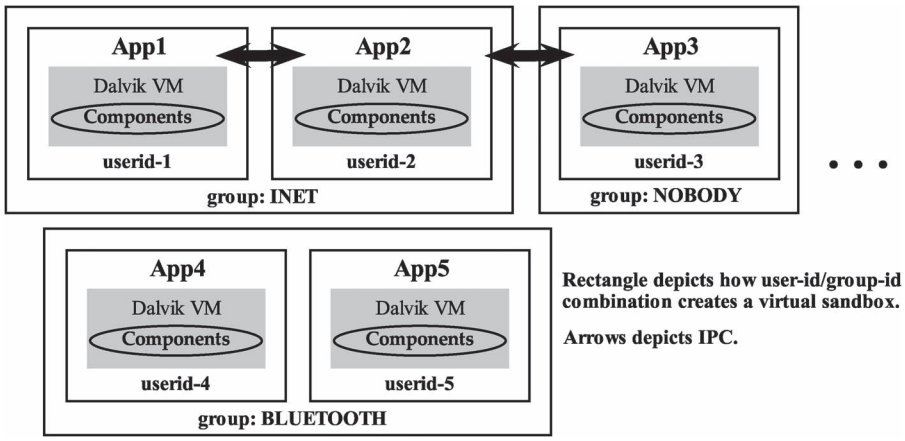


Fig. 4. Android Apps within Sandbox at Kernel-level [34].

app into an isolated sandbox assigning it a unique UID. If the certificate of an app A matches with an already installed app B on the device, Android assigns the same UID (i.e., sandbox) to apps A and B, permitting them to share their private files and the manifest defined permissions. This unintended sharing can be exploited by the malware writers as naive developers may generate two certificates. It is advisable for the developers to keep their certificates private to avoid their misuse.

E. Permissions at Framework-Level

To restrict an app from accessing the sensitive functionality such as telephony, network, contacts/SMS/sdcard and GPS location, Android provides permission-based security model in the application framework. Developer must declare the permissions required using the ⟨`uses-permissions`⟩ tag in `AndroidManifest.xml` as discussed before in the Listing 1 at line number two, four and six respectively. Android controls the individual apps to mitigate the undesirable effects on the system apps or third party developer apps within the sandbox. These restrictions are enforced on the process at the install time. Android permissions are divided into the following four protection-levels [37]:

1) *Normal*: These permissions have a minimal risk on the user, system app or the device. Normal permissions are granted by default at the install time.
2) *Dangerous*: These permissions fall within the high risk group due their capability of accessing the private data and important sensors of the device. A user must accept the installation of dangerous permissions at the install time.
3) *Signature*: These permissions are granted only if the requesting app is signed with the same developer certificate of the app that declared the permissions. They are granted automatically at the install time. Signature permissions are available with the system apps.
4) *SignatureOrSystem*: These permissions are granted if the requesting app is signed with the same certificate as the Android system image or with an app that declared this permissions. They are granted automatically at installation time.

Android permissions are coarse-grained. For example, the `INTERNET` permission does not have the capability to restrict access to a particular Uniform Resource Locator (URL). `READ_PHONE_STATE` allows an app to identify whether the device rings or is on hold. At the same time it also allows the app to read the sensitive information such as device identifiers. Permissions such as `WRITE_SETTINGS`, `CAMERA` are broadly defined, thus it violates the least privilege access principle. Access to `WRITE_CONTACTS` or `WRITE_SMS` does not imply the access to `READ_CONTACTS` or `READ_SMS` permissions. Thus permissions are not hierarchical and they must be separately requested by the developer. At the install time, the user is forced to grant either all permissions or deny the app installation. Hence the dangerous permissions cannot be avoided at the install time. Moreover, the users cannot differentiate between the necessity and its imperative misuse which may expose the for exploitation [38].

F. Secure System Partition

Android system partition is built from the kernel, system libraries, the android runtime, app framework and the apps [34]. Android system partition are read-only to protect the unauthorized access and/or modifications. Also, some part of file-system such as application cache and sdcard are protected with the appropriate privileges to prevent its tampering by the adversary when the device is connected to the desktop through the USB.

G. Secure Google Play Store

Google discourages the users to install apps to thwart any third party market place app due to the security concern. However, it still permits the installation from other third party markets. Third party developer apps are made available from the official playstore. Google vets the third party developer app with Bouncer [6], a dynamic analysis sandboxed environment to thwart any malware from entering the Google Play. Bouncer, if not invincible is a reasonably effective security mechanism [39]. Android has the facility of running a verification service while installing apps from other market places. Google Play is capable of remote un-install if it finds the malicious behavior [40]. However, this facility is available for the devices connected to the Internet.

III. Android Security Issues and Enhancements

This section gives a detailed description about the user and device security issues. Moreover, it covers various enhancements employed by the AOSP in subsequent Android versions.

A. Android Threats

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 [41]. Android exported components 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 [42].
- *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 functionality 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.
- *Denial of Service (DoS)* attack can happen when app(s) overuses already limited CPU, memory, battery and bandwidth resources and restrains the users executing normal functions.

B. Version Update Issues

Android Open Source Project (AOSP), led by Google, upgrades and maintains Android source-code. However, the patch, an update or major upgrade distribution release remains the responsibility of Original Equipment Manufacturers (OEMs) or the wireless carriers. Individual OEM branches out updated versions of the OS and customize them accordingly. In some countries, the wireless carriers customize the OEM OS to suit their own requirements. Such an update chain takes months before the patch reach the end-users. This phenomenon is called *Fragmentation*, where different versions of Android remain scattered due to unavailability of updates. Specifically, handsets with older and un-patched versions remain vulnerable to the known exploits.

Android OS updates and upgrades are more frequent compared to the desktop OS. Android has released 29 stable OS version updates and upgrades since its launch in September 2008 [43]. Over The Air (OTA) update significantly changes the existing version modifying the large number of files across the platform, maintaining the integrity of existing user data and apps [44]. New version update is facilitated through a service called Package Management System (PMS). Xing *et al.* [44] performed a comprehensive pileup vulnerabilities study which in turn can be exploited by the malware authors during the version upgrades. An app developed for the older version can be exploited to use the dangerous permission(s) introduced in the higher version release. During the update, Android does not verify the appended permissions in the updated app [44]. Thus, it compromises the device security. During a major update or upgrade, large number of files are modified ensuring the sensitive user information remains intact leading to complexity in update procedures.

C. Native Code Execution

Android allows native code execution through libraries implemented in C/C++ using Native Development Kit (NDK). Even though native code executes outside Dalvik VM, it is sandboxed through user-id/group-id(s) combination. However, native code has the potential to perform privilege escalation by exploiting platform vulnerabilities [23], [45]–[49], demonstrated by quite a few malware attacks in the recent past [50].

D. Security Enhancements in the Recent Versions

In the view of security issues, vulnerabilities and/or reported malware attacks, AOSP releases patches, updates, enhancements and upgrades. Here, we discuss notable security fixes and features incorporated in the subsequent Android OS versions up to Android Kitkat 4.4:

1) Android prevented stack buffer and integer overflow in the OS version 1.5. In version 2.3, Android fixed string format vulnerabilities, and added hardware based No eXecute (NX) support to stop execution of code in stack and heap [34].
2) In Android 4.0 Address Space Layout Randomization (ASLR) was added to prevent the *return-to-libc and memory related attacks* [34].
3) Information can be ex-filtrated by connecting the device to a PC using the Android Debug Bridge (ADB) driver. Though the ADB is developed as a debugging tool, it permits app installation/install/un-install, reading system partitions etc. even if the device is locked, but connected to a Personal Computer (PC). To prevent such unauthorized access, Android 4.2.2 authenticates an ADB connection using RSA keypair [51]. User response is

prompted on the device screen if the ADB connection accesses the device. Thus, if the device is locked, attacker would not be able to gain the control.

4) To prevent the malware from silently sending premium-rate SMS messages, Android 4.2 introduced an additional notification feature to prompt the user before a user app sends an SMS [52].
5) Android introduced a major capability addition to the version 4.2 (API version 17) permitting creation of multiple users (MU) to allow multiple users access a shared device such as tablet [53]. Restricted profile (RP) access capability was introduced added in Android 4.3 (API version 18) in July 2013. These modifications were placed keeping in mind the usage of sharable mobile devices such as tablets to provide private space to multiple users on a single mobile device. For each user, a separate account, user selected apps, custom settings, private files and private user data is assigned. This capability enables the multiple users share a single device. In the MU scenario, main account is the owner of the device. Using device settings, owner can create additional MUs. Except the original user, other created MU user cannot create, modify or delete the device MU users.
6) Android 4.3 removed the `setuid()/setgid()` programs [51] as they were vulnerable to the root exploits.
7) Android 4.3 experimented with SELinux to provide the enhanced security [54]. Android 4.4 introduced SELinux with enforcing mode for multiple root processes. SELinux imposed Mandatory Access Control (MAC) policies in place of the traditional Discretionary Access Control (DAC). In DAC, the owner of the resource decides which other interested subjects can access it, where as in MAC the system (not the users) authorizes the subject to access a particular resource. Thus, MAC has the potential to prevent the malicious activity(s) even if the root access of the device is compromised. Thus, MAC substantially reduces the effect of kernel-level privilege escalation attacks.

E. Third-Party Security Enhancements

Many independent Android security enhancements have been proposed [55]–[58]. These mechanisms allow an organization to create fine grained security policies for their employee devices. Contextual information such as device location, app permissions and inter-app communication can be monitored and verified against the already declared policies. Scope of this paper is to investigate Android security, malware issues and defense techniques, it does not examine the above mentioned prevention techniques in detail.

IV. Reported Android Malware Threat Perception

Fig. 5 illustrates the time-line of some notable malware families of Android during 2010–2013. Among them, SMS Trojans have major contribution; some of these have even infected the Google Playstore [50]. A large number of malware apps have exploited root-based attacks such as *rage-against-the-cage* [23], *gingerbreak* [48] and *z4root* [45] to gain superuser privileges to control the device. The most recent android exploit is the *master-key* attack [59], which has the versions starting from 1.6 to 4.2.2 vulnerable.

Each quarter, the anti-malware companies report an exponential increase in the new families and existing malware variants [3], [60]. These companies differ in the approximation of the malware infection-rate on Android devices. In particular, Lookout Inc. reported the global malware infection-rate likelihood percentage 2.61% for its users [61]. Two independent researches estimated the real infection-rate. 1) In [62], the authors used the smartphone Domain Name Resolution (DNS) traffic in the United States and reported 0.0009% infection. 2) Truong *et al.* [63] instrumented the Carat app [64] to estimate the infection-rates for three different malware datasets reporting 0.26% and 0.28% for McAfee and Mobile Sandbox dataset respectively. Thus, the present Android threat perception and malware infection rate has a huge reported variation between the commercial anti-malware and independent studies.

In the following paragraph, we discuss the Android malware classified and its characteristics.

A. Trojan

Trojans masquerade as benign apps, but they perform harmful activities without consent or knowledge of the users. Trojans leak the confidential user information, or they may "phish" the user and steal the sensitive information such as passwords. Till the second quarter of 2012, majority of the android variants belonged to various SMS trojan families. SMS trojan apps are capable of sending SMS to premium rate numbers without the knowledge and/or consent of the user incurring financial loss to the owner. Apart from that, such trojans also divulge contacts, messages, IMEI/IMSI numbers to the command and control domains. *FakeNetflix* [65] masquerades itself as popular Netflix app, phishing the user to enter their login credentials. *Fakeplayer* [42], *Zsone* [3] and *Android.Foney* [66] are a few notable Android trojans incurring financial loss to the user.

On account of the increased mobile banking transactions, malware authors have targeted the two-factor mobile banking authentication. After capturing the username and password of target accounts employing social engineering attacks, Zitmo and Spitmo Trojans monitor and steal the mTANs (Mobile Transaction Authentication Numbers) to silently complete transactions [67].

B. Backdoor

Backdoor allows other malware to silently enter the system facilitating them the bypass of the normal security procedures. Backdoor can employ root exploits to gain the superuser privilege and hide from the anti-malware scanners. A number of root exploits such as *rage-against-the-cage, rageagainstthecage* and *gingerbreak* [48] gain full-control of the device. *Basebridge* [50], *KMin* [50], *Obad* [22] are notable example of the known backdoors.

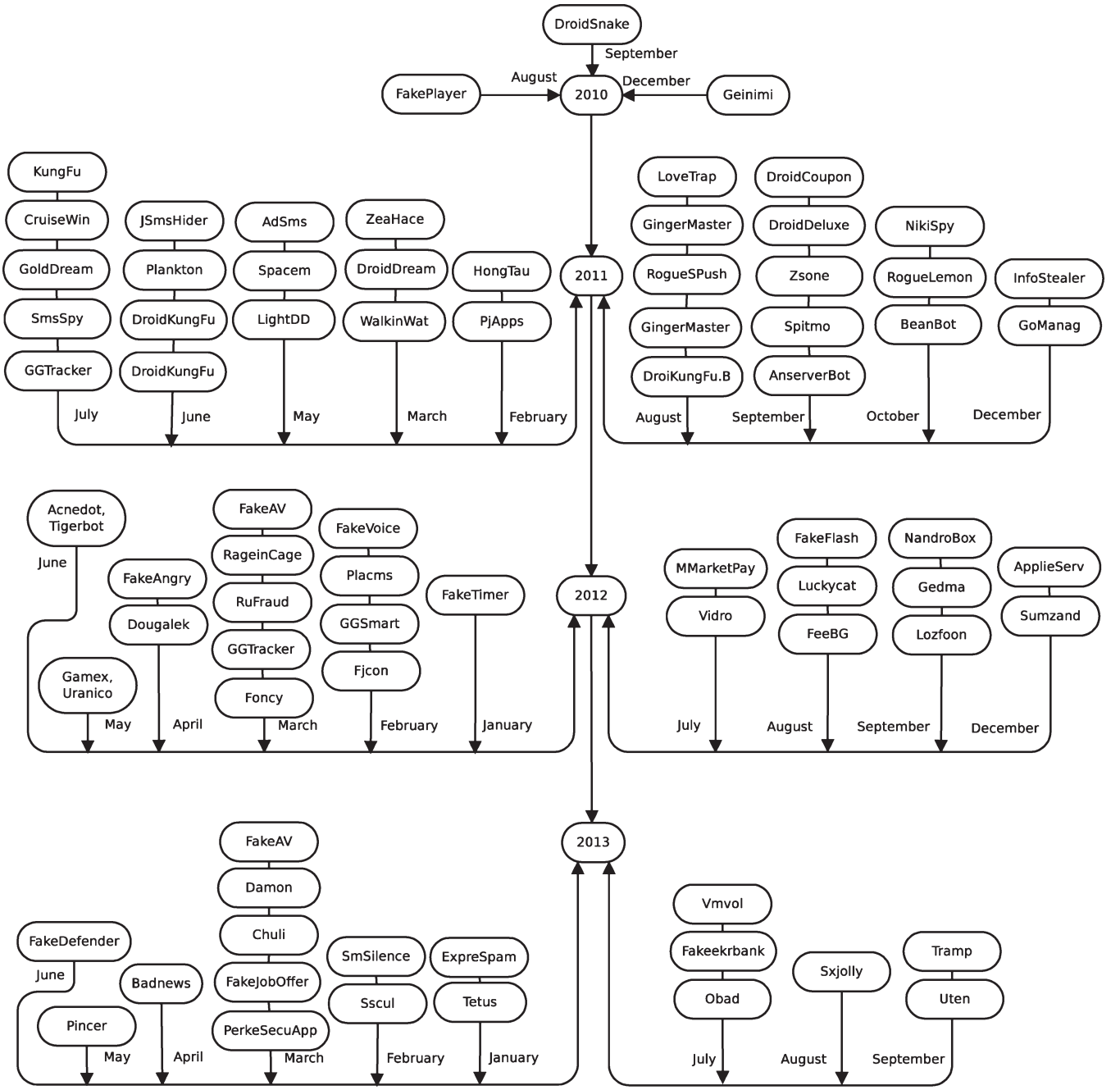


Fig. 5. Android Malware Family Chronology [2], [14]–[16], [18], [19].

C. Worm

Worm app can create an exact or similar copies of itself and spreads them through network and/or removable media. For example, Bluetooth worms can exploit bluetooth functionality and send copies to the paired devices. *Android.Obad.OS* [22] is well known bluetooth worm.

D. Botnet

Botnet apps compromise the device to create a *Bot*, so that the device is controlled by a remote server, called *Bot-master*, through a series of commands. Network of such *bots* is called a *Botnet*. Commands can be as simple as sending private information to remote-server or as complex leading to denial of service attacks. *Bot* can also include commands to download malicious payloads automatically. *Geinimi* [50], *Anserverbot* [50], *Beanbot* [50] are notable Android botnets.

E. Spyware

Spyware may present itself as a good utility, but has a hidden agenda to surreptitiously monitor contacts, messages, location, bank mTANs etc. that leads to undesirable consequences. It can send the collected information to the remote server. *Nickyspy* [50], *GPSSpy* [3] are known examples of spyware apps.

F. Aggressive Adware

Android provides coarse and fine grained location services. Some advertisement affiliate networks misuse such location services and send personalized advertisements to the user device to generate revenues. Aggressive adware can create shortcuts on the home-screen, steal bookmarks, change the default search engine settings and pushing unnecessary notifications to hinder the effective device usage. *Plankton* [3] is a known aggressive adware.

G. Ransomware

Ransomware can lock the user device to make it inaccessible until some ransom amount is paid through online payment service. For example, *FakeDefender.B* [68] masquerade as *avast!* [69] anti-malware and displays the fake malware alerts to coax the user install this hoax malware. In addition, it locks the device and demands ransom to unlock the device.

V. Malware Penetration and Survival Techniques

In this section, we summarize the malware penetration and state of the art stealth techniques employed by the Android malware apps.

A. Repackaging Popular Apps

Repackaging is a process of disassembling/decompiling the popular free/paid apps from the popular market places, insert, append 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 Fig. 6. The following section discusses the main steps involved in app repackaging:

- Download the popular free/paid app from the popular app-store(s).
- Disassemble the app with a disassembler such as *apktool* [70].
- Generate a malicious payload in dalvik bytecode or Java and convert it to the bytecode using the *dx* [71] tool.
- Add the malware payload into benign app. Modify the `AndroidManifest.xml` and/or *resources* if required.
- Assemble modified source again using *apktool*.
- Distribute repackaged app by self-signing with another certificate to the less monitored third party app market.

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. Malware authors can divert advertisement revenues by replacing the advertisements of the original developers.

AndroRAT APK Binder [72] repackages and generates a trojanized version of a popular and legitimate equipping it with the Remote Access functionality. Adversary can remotely force the infected device to send SMS messages, make voice calls, access the device location, record video and/or audio and access the device files using the remote access service.

B. Drive-by Download

An attacker can employ social engineering, aggressive advertisements and click a malicious URL, incite user to download malware automatically. Optionally, a drive by download may disguise a legitimate application and coax the user install an app. *Android/NotCompatible* [25] is a notable drive-by download app.

C. Dynamic Payload

An app can also embed malicious payload as an executable `apk`/`jar` in encrypted or plain format within the APK resources. Once installed, the app decrypts the payload. If the payload is a `jar` file, malware loads `DexClassLoader` API and execute dynamic code. However, it can coax the user install the embedded `apk` by disguising as an important update. The app can execute native binaries using `Runtime.exec` API, an equivalent of Linux `fork()`/`exec()`. *BaseBridge* [50] and *Anserverbot* [50] malware families employs the above discussed technique. Some malware families does not embed malicious payload as a resource, but rather download them from the remote server and successfully evade detection. *DroidKungFuUpdate* [50] is a notable example of dynamically executing payload. Such techniques go undetected with static analysis methods.

D. Stealth Malware Techniques

Android OS is developed for resource constrained environment keeping in mind the availability of limited battery availability of the underlying smartphone. On device anti-malware apps cannot perform the real-time deep analysis unlike their desktop counterpart. Malware authors exploit these hardware constraints limiting the anti-malware and obfuscate the malicious payloads to thwart the commercial anti-malware. Stealth techniques such as code encryption, key permutations, dynamic loading, reflection code and native code execution remain a matter of concern for signature-based anti-malware solutions.

Following the trends of the desktop platform, code obfuscation is also evolving on Android [73], [74]. Obfuscation techniques are implemented for one or more of the following purposes.

- To protect the proprietary algorithm from rivals by making the reverse-engineering difficult.
- To protect Digital Rights Management of multimedia resources to reduce piracy.
- Obfuscating the apps make them compact and thus faster in execution.

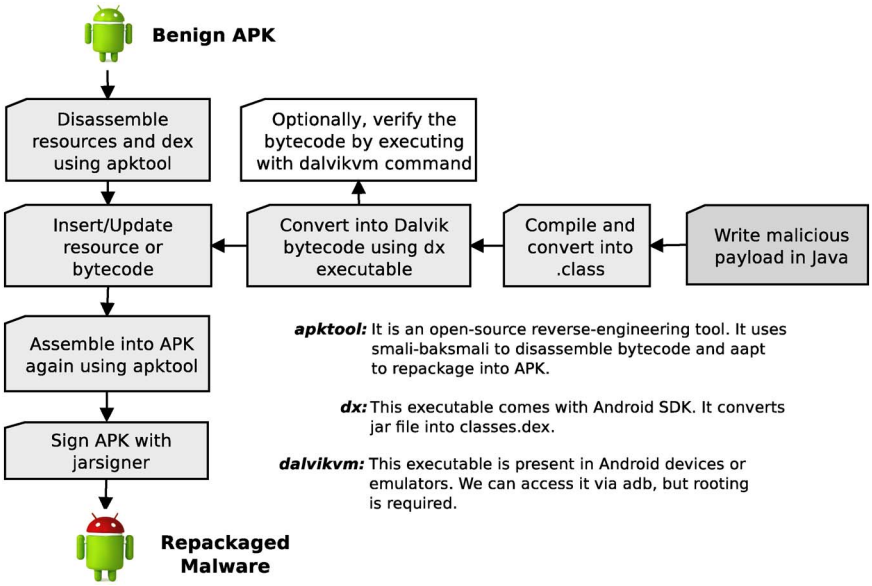


Fig. 6. App Repackaging Process.

- To hide the already known malware from anti-malware scanners to propagate and infect more devices.
- To prevent or at least delay human analysts and/or automatic analysis engines from figuring out actual motive of the unknown malware.

Dalvik bytecode is amenable to reverse-engineering due to the availability of type safe information such as class/method types, definitions, variables, registers literal strings and instructions. Code transformation methods can be easily implemented on dalvik bytecode, optimize it with a code protection tool such as *Proguard* [75]. *Proguard* is an optimization tool to remove the unused classes, methods and fields. Meaningful class/method/fields/local-variable names are replaced with unreadable code to harden the reverse engineering. *Dexguard* [76] is a commercial Android code protection tool. It can be used to implement code obfuscation techniques such as class encryption, method merging, string encryption, control flow mangling etc. to protect app from being reverse-engineered. Code transformation techniques can also be used to hinder the malware detection approaches [73], [74]. Faruki *et al.* [77] proposed an automated dalvik bytecode transformation framework to generate unseen variants of already known malware with different bytecode obfuscation techniques. In addition, they also evaluated the unseen malware samples against the top commercial anti-malware and static analysis techniques. The authors reported that, even trivial transformation techniques can fail the existing commercial anti-malware.

In the following, we cover various code transformation methods used to obfuscate the existing known malware and generate huge number of unseen malware signatures. In fact, code transformation can also implemented to thwart the disassembly tools [78].

1) Junk Code Insertion and Opcode Reordering: Junk code or *no-operation* code (nop) insertion is a well-known technique that changes the executable size and evades the anti-malware signature database. Junk code insertion preserves the semantics of the original app. However, it changes the opcode sequence to alter the malware app signature. Opcode can be re-ordered with the *goto* instructions in-between the functions and alter the control flow, preserving the original execution semantics. These methods can be used to evade the signature-based or opcode-based detection solutions [73], [74].

2) Package, Class or Method Renaming: Android app is uniquely identified with its unique package name. Dalvik bytecode being type safe preserves the class and method names. Many anti-malware use trivial signatures such as package, class or method names of a known malware as detection signature [79]. Such trivial transformations can be used to evade the anti-malware signature based detection [74].

3) Altering Control-Flow: Some anti-malware use semantic signatures such as control flow and/or data flow analysis to detect the malware variants employing simple transformation techniques [79]. Control flow of a program can be modified with the *goto* instructions or by inserting and calling the junk methods. Though trivial, such techniques evade the commercial anti-malware [74].

4) String Encryption: Literal strings like messages, URLs and shell-commands reveal a lot about the app. To prevent such analysis, the plain text strings can be encrypted and made unreadable. Also, each time the string encryption is executed, various encryption methods (or keys) make it difficult to automate the decryption process. In that case, literal strings can only be available during the code execution. Hence, it evades the static analysis methods.

5) Class Encryption: Important information such as product license-checks, paid downloads and DRM can be hidden by encrypting the entire classes utilizing the above sensitive information [76].

6) Resource Encryption: Content of *resources* folder, *assets* and native libraries can be altered as unreadable, hence they must be decrypted at runtime [76].

7) Using Reflection APIs: Static analysis methods search sensitive Android API within the malware apps map the malicious behavior. User apps permits Java reflection allowing

the creation of programmatic class instances and/or method invocation using the literal strings. To identify the exact class or method names, data-flow analysis can be implemented. However, the literal strings can be encrypted, making it hard to automatically search the reflection API. Such techniques can easily evade static analysis approaches.

VI. Approaches for Assessment, Analysis, and Detection

Android security solutions such as vulnerability assessment, malware analysis and detection techniques are divided into: 1) Static; 2) Dynamic and 3) Hybrid. Static analysis methods analyze code without actually running it, hence they are quick, but they have to deal with false-positives. Dynamic analysis techniques monitor the executed code and inspect its interaction with the system. Though time-consuming they are effective against malware obfuscation. Hybrid approaches leverage the good of both the static and dynamic analysis methods.

Security solutions can be categorized as rule-based [80] or feature extraction based machine-learning models [81]. Inappropriate feature selection can degrade the performance of model, to generate false-positives (i.e., false detection of benign apps as malware). Moreover, the number of features under the problem must be small sized and effective as an on device anti-malware solution. Feature reduction methods combined with statistical measures such as *mean, standard deviation, chi-square, haar transforms* can be used to identify the prominent attributes responsible for malicious actions. Learning models can be created by analyzing the features such as processor, memory usage, battery consumption, system call invocation, network activity etc. that can be used with the clustering or classification algorithms to predict anomalous behavior.

A. Static Approach

Static analysis based approaches work by just disassembly, decompilation without actually running it, hence does not infect the device. This approach is undermined by the use of various code transformation techniques discussed in this review in Section V-D.

1) Signature-Based Malware Detection: The existing commercial anti-malware use signature based malware detection approaches. It extracts the interesting syntactic or semantic patterns, features [82] and create a unique signature matching that particular malware. Signature-based methods fails against the unseen variants of already existing and known malware. Moreover, the signature extraction process being manual, its efficacy in the wake of exponential unique signature outbreak may leave the device vulnerable to malware attacks. Faruki *et al.* [83] developed AndroSimilar, an automated robust statistical feature signature based method to detect zero-day variants of the already known malware.

2) Component-Based Analysis: In order to perform detailed app-security assessment or analysis, an app can be disassembled to extract the important content such as `AndroidManifest.xml`, resources and bytecode. Manifest stores important meta-data about such as list of the components (i.e., activities, services, receivers etc.) and required permissions. App-security and assessment solutions can analyze the components using their definition and bytecode interaction to identify the vulnerabilities [8], [84], [85].

3) Permission-Based Analysis: Requesting permission to access a sensitive resource is the central design of Android security model. No application by default has any permission that can affects user security. Identifying the dangerous permission request is not sufficient to declare the malware app, but nevertheless, permissions mapping requested and used permissions is an important risk identification technique [86], [87].

Sanz Borja *et al.* [38] used $\langle$`uses-permission`$\rangle$ and $\langle$`uses-features`$\rangle$ tags present in `AndroidManifest.xml` to detect malware apps. Authors utilized machine learning algorithms Naive Bayes, Random Forest, J48 and Bayes-Net on a dataset of 249 malware and 357 benign apps. In [88] authors mapped the requested and used permissions from the manifest and their corresponding API in the dalvik bytecode. The mapped attributes were used with the machine learning algorithms on 125,249 malware and benign app dataset. Enck *et al.* [89] developed a certification tool, *Kirin* to define a set of rules to identify the combination of specific dangerous permissions to identify malware attributes before installing the app on device.

4) Dalvik Bytecode Analysis: Dalvik bytecode is semantically rich containing type information such as classes, methods and instructions. The type information can be utilized to verify the app behavior. Detailed analysis based on control and data flow gives an insight into the dangerous functionality such as privacy leakage and telephony services misuse [30], [80], [90]. Control and data flow analysis are also useful to rebuild a de-obfuscated bytecode, for example and nullify the effect of trivial transformation techniques [91].

Bytecode control-flow analysis identifies the possible paths that an application can take while it is executed. Dalvik bytecode contains jump, branch and method invocation instructions that alter execution order. To facilitate further analysis, an intra-procedural (i.e., within a single method) or inter-procedural (i.e., spanning across methods) control-flow bytecode graph (CFG) is generated. Karlsen *et al.* [91] formalized the Dalvik bytecode to perform the control-flow analysis based semantic signatures to detect malware apps.

Bytecode data-flow analysis predicts the possible set of values during the different point of execution. CFG can be used to traverse the possible execution paths to determine the control and data dependency. Data-flow analysis is performed within methods (intra-procedural) or between different methods (inter-procedural level) to improve the approximation of the desired output. In particular, special data-flow analysis also known as "constant propagation" is implemented to identify the constant arguments of sensitive API calls invoked during the app execution. For example, a malware app sending premium rate SMS to a pre-defined hard coded number can be detected with the constant propagation data-flow analysis [80]. Taint analysis another type of data-flow analysis method to identify the colored variables holding the sensitive information. For example, taint analysis can identify privacy leakage which can be used to steal the sensitive user information apps [90]. Sensitive API-call tracking within the bytecode can be useful to identify

malicious behavior [92]. It is also helpful in identifying the app clones [93]. Zhou *et al.* [4] utilized the sequence of opcodes in the Dalvik bytecode instructions to identify the repackaged malware apps.

5) Re-Targeting Dalvik Bytecode to Java Bytecode: Availability of number of Java decompilers [94]–[96] and static analysis tools based on [97]–[99], has motivated the researchers to re-target the Dalvik bytecode to the Java bytecode. Enck *et al.* [100] developed the *ded* tool that is used to convert Dalvik bytecode to Java source. Later, they performed static analysis control-flow, data-flow, on the Java code using Fortify SCA [99] framework. In [101] authors developed *Dare* tool to convert the Dalvik bytecode to Java bytecode with 99% accuracy. Bartel *et al.* [102] developed the *Dexpler* plugin for static analysis framework, *Soot* [97]. *Dexpler* converts the Dalvik bytecode into Soot's internal Jimple code. However, it is unable to handle the optimized *dex* (odex) files. Gibler *et al.* [103] employed *ded* and *dex2jar* [104] to convert the Dalvik bytecode into Java bytecode and source code respectively. Authors implemented static analysis WALA [98] framework to identify the privacy leakage within Android apps on a fairly big dataset.

B. Dynamic Approach

Static analysis and detection approaches are quick, they fail against the encrypted, polymorphic and code transformed malware. Dynamic analysis methods execute the app in a protected environment, providing all the emulated resources it needs, thereby learning its interaction identify malicious activities. Some dynamic analysis methods have been implemented, but the resource constraints of a smartphone limits such execution methods. Android app execution being event based with asynchronous multiple entry points, it is important to trigger those events. User Interface (UI) gestures such as tap, pinch, swipe, keyboard and back/menu key press must be automatically triggered to initiate the app interaction with the device. Android SDK comes is equipped with the *monkey* [105] tool, to automate some of the above gestures discussed above. In order to perform an in-depth monitoring, one may need to modify the framework by inserting the tracking code known as *Instrumentation*.

A serious drawback of dynamic approach is that some malicious execution path may get missed, if it is triggered according to some non-trivial event. For example, at a particular time of the day the malware functionality is executed, but that event is never executed. Anti-emulation techniques such as Sandbox [39], [106] detection, timing out the analysis environment, delaying the malware execution can evade the dynamic analysis methods. Dynamic approaches are divided into the following three categories.

1) Profile-Based Anomaly Detection: Malicious apps may create Denial of Service (DoS) attacks by over utilizing the constrained hardware resources. Range of parameters such as CPU usage, memory utilization statistics, network traffic pattern, battery usage and system-calls for benign and malware apps are collected from the Android subsystem. Automatic analysis techniques along with machine learning methods are used to distinguish the abnormal behavior [81], [107], [108].

2) Malicious Behavior Detection: Specific malicious behaviors like sensitive data leakage, sending SMS/emails, voice calls without user consent can be accurately detected by monitoring the particular features of interest [109]–[112].

3) Virtual Machine Introspection: The downside of app behavior monitoring from an emulator (VM) is, an emulator itself is susceptible against the malicious app which defeats the analysis purpose. To counter this, Virtual Machine Introspection approaches can be employed to detect app behavior by observing the activities out of the emulator [113].

VII. Deployment for Assessment, Analysis, and Detection Approaches

Security assessment, malware analysis and detection methods can be deployed at different places, depending on the requirement, from on-device solution to a completely off-device or cloud base techniques.

A. On-Device

Signature-based malware is simple and efficient. The detailed assessment and analysis remains constrained on a mobile as compared to the desktop anti-malware analysis. Thus, lightweight risk assessment solutions can be proposed by analyzing the components and permissions as an on device solution [89]. Following are some on device anti-malware limitations.

- Anti-malware apps run as a normal app without any special privileges. As a result, they are also under the purview of process isolation. Hence, they cannot directly scan other app memory, files read/written and private files during the app scanning.
- Android permits execution of background app services. However, it can stop anti-malware app services if it runs out of hardware resources. Similar privileged apps can force stop an anti-malware app execution with appropriate privileges.
- Without acquiring the *root* privileges, anti-malware app cannot create system hooks to monitor the file-system or perform network access.
- Without acquiring *root* privileges, anti-malware app cannot uninstall any other app. It has to depend upon the user for removing the app.

B. Distributed (Some Part On-Device, Some Part Off-Device)

On the fly analysis and/or detection can be performed on the device, detailed and computationally expensive analysis can be performed at remote server to make anti-malware app limited-resource friendly. In the case of profile-based anomaly detection, resource usage parameters are be collected at the client-side and sent back to the remote server for detailed analysis. The results can be finally sent back to the device [81], [110]. However, continuous availability of the Internet bandwidth and associated cost is a concern. In case of unavailability of network

resources host-based detection approach can protect the device from malware attack [108].

C. Off-Device

It is important to automate the deep static analysis of a new malware sample to enable the human analysts take quick decision to identify and mitigate the malware. Such automated deep analysis solutions need computational power and memory. Due to this, they are usually deployed off-device [30], [80], [90], [113].

VIII. State-of-the-Art Tools & Techniques for Android App Assessment, Analysis, and Detection

Industry and academia have proposed several solutions for Android malware analysis and detection. In this section, we survey and examine promising reverse-engineering tools and detection approaches. Detection approaches have been classified according to the following: 1) Goal, which can be app-security assessment, analysis and/or malwaredetection; 2) Methodology as discussed in Section VI; and 3) Deployment discussed in Section VII.

A. Reverse-Engineering Tools

Content of Android package (APK) is stored in the binary format. Before assessment, analysis or detection task initiates, it is important to disassemble it for further processing. There are a number of tools to disassemble and/or decompile the Android app. In the following section, we discuss some known reverse-engineering tools considering their strengths.

1) *apktool* [70] can decode binary content of an APK into nearly original form in project-like directory structure. It disassembles the binary resources and converts bytecode within `classes.dex` into the smali [114] bytecode for easier reading and manipulation. After making the changes, it can also repackage it back into an APK. This tool is one of the best open source reverse-engineering tool.
2) *dex2jar* [104] is a disassembler to parse both the `.dex` and optimized `dex` file, providing a light-weight API to access it. *dex2jar* can also convert `dex` to a `jar` file, by re-targeting the Dalvik bytecode into Java bytecode, for further manipulation. Moreover, it can also re-assemble the `jar` into a `.dex` after the modifications.
3) *Dare* [115] project aims at re-targeting Dalvik bytecode within `classes.dex` to traditional `.class` files using strong type inference algorithm. This `.class` files can be further analyzed using a range of traditional techniques developed for Java applications, including the decompilers. Octeau *et al.* [101] demonstrated that *Dare* is 40% more accurate than *dex2jar*.
4) *Dedexer* [116] disassembles the `classes.dex` into Jasmin-like syntax and creates a separate file for each class maintaining the package directory structure for easy reading and manipulation. However, unlike the *apktool*, it cannot re-assemble the dis-assembled intermediate class files.
5) *JEB* [117] is a leading professional Android reverse-engineering software available on Windows, Linux and Macintosh platforms. It is a GUI-based interactive decompiler to analyze the reversed malware app content. App information such as manifest, resources, certificates, literal strings can be examined in Java source by providing an easy navigation through the cross-references. JEB converts the Dalvik bytecode directly into Java source by utilizing dalvik bytecode semantics. Exceptionally, JEB can also de-obfuscate Dalvik bytecode to make disassembled code more readable in comparison to its counterparts [70], [104]. JEB supports Python scripts or plugins by allowing access to the decompiled Java code Abstract Syntax Tree (AST) through API. This feature is helpful in automating the custom analysis. According to us, it is the best reverse-engineering tool so far.

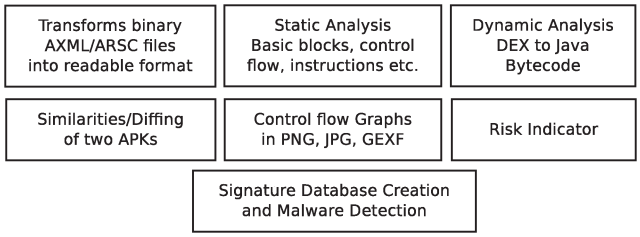


Fig. 7. Features of Androguard.

B. Androguard

Goal: Risk Assessment, Analysis and Detection
Methodology: Static
Deployment: Off-Device

Illustrated in Fig. 7, Androguard [79] an open-source, static analysis tool can reverse engineer to disassemble and decompile Android apps. It generates the control flow graphs for each method and provides access through Python-API on the command line and graphic interface. Androguard Normalized Compression Distance (NCD) approach finds similarities and differences of two suspected clones reliably, which is also helpful to detect repackaged apps. It provides python APIs to access the disassembled resources and static analysis structures like basic-blocks, control-flow and instructions of an APK. An analyst can develop his own static analysis framework using the python APIs. Following are some of the features explained below.

1) App Code Similarity: Androguard finds similarities between two apps by calculating Normalized Compression Distance between each method pairs and calculates a similarity score between 0–100, where 100 means identical apps. It displays IDENTICAL, SIMILAR, NEW, DELETED and SKIPPED methods of the two suspected clones. In the same way, it displays differences between two methods by comparing each basic blocks pairs. More specifically, to calculate differences between two similar methods, it first converts each unique instruction in basic block into a string. Then, it applies Longest

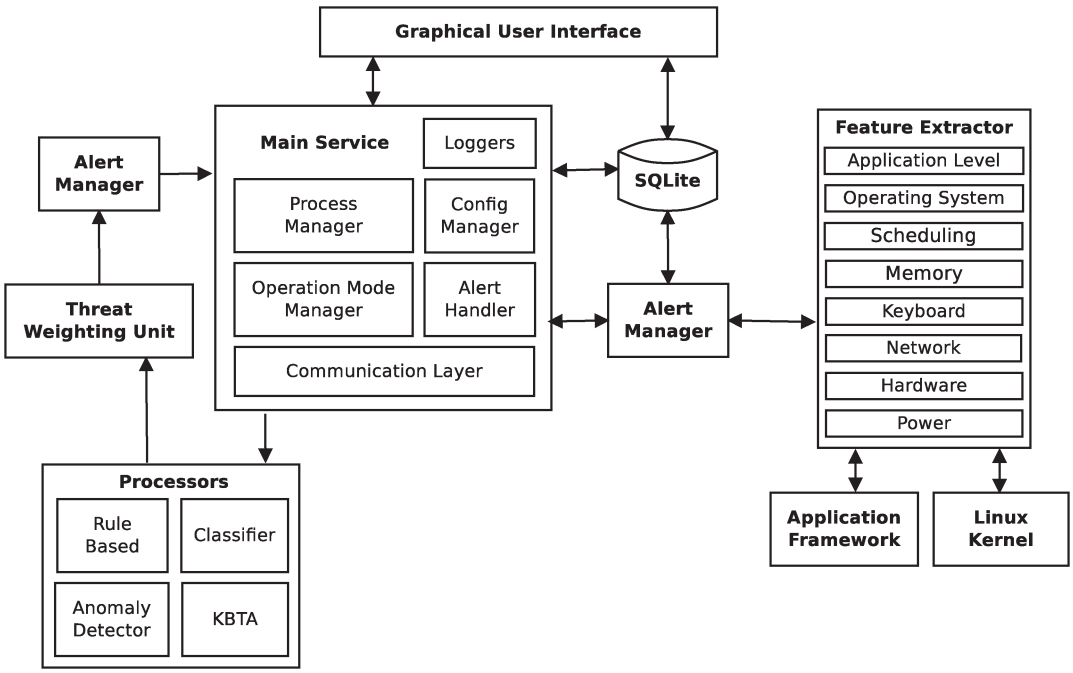


Fig. 8. Architecture of Andromaly.

Common Subsequence algorithm on these strings of two basic blocks to find differences between them [118].

2) Risk Indicator: Risk Indicator calculates fuzzy risk score of an APK from 0 (low risk) to 100 (high risk). It considers following parameters:

- Native, Reflection, Cryptographic and Dynamic code presence in an app.
- Number of executables/shared-libraries present in an app.
- Permission requests related to privacy and monetary risks.
- Other *Dangerous/SystemOrSignature/Signature* permission requests.

3) Signature of Malicious Apps: Androguard manages a database of signatures and provides an interface to add/remove signatures to/from the database. Signature is described in the JSON format. It contains a name (or family-name), set of sub-signatures and a Boolean formula to mix different sub-signatures. Following are the two types of sub-signatures:

- METHSIM: It contains three parameters, CN—class name, MN—method name and D—descriptor.
- CLASSSIM: It contains a single parameter, CN—class name.

Thus sub-signature can be applied on a specific method or entire class. Different sub-signatures can be mixed with Boolean formula (BF).

C. Andromaly

Goal: Anomaly Detection
Methodology: Dynamic
Deployment: Half On-Device, Half Off-Device

In [81], Shabtai *et al.* have proposed a light-weight Android malware detection system based on machine learning approach. It performs real-time monitoring for collection of various system metrics, such as CPU usage, amount of data transferred through network, number of active processes and battery usage.

As shown in Fig. 8, Andromaly has four major components:

- *Feature Extractors*: They collect feature metrics, by communicating with Android kernel and application framework. Feature Extractors are triggered at regular intervals to collect new feature measurements by the feature manager. Feature Manager may also perform some pre-processing on the raw feature data.
- *Processor*: It is an analysis and detection unit. It receives the feature vectors from Main Service, analyze them and perform threat assessment and pass it on to Threat Weighting Unit (TWU). Processors can be rule-based, knowledge-based classifiers or anomaly detectors employing machine learning methods. TWU applies ensemble algorithm on the analysis results received from all the processors to derive a final decision on the device infection. Alert Manager smoothes the results to reduce the false alarms.
- *Main Service*: It coordinates feature collection, malware detection and alert process. It is responsible for requesting new feature measurements, sending new feature metrics to the processors and receives final recommendations from the alert manager. Loggers can log information for debugging, calibration and experimentation. Configuration Manager manages the configuration of an application, for example, active processors, alert threshold, sampling interval etc. The task of activating or deactivating processors is taken care by Processor Manager. Operation Mode Manager switches application from one mode to another that results in the activation/deactivation of processors and feature extractors. This change in operation modes is resulted due to change in resource levels.

Fig. 9. AndroSimilar Methodology.

- *Graphical User Interface*: It interacts with the user to configure application parameters, activate/deactivate the application, alerts user regarding threats and allows exploring collected data. Experiments were carried out using few categories of artificial malware, thus working model needs testing by real malware.

D. AndroSimilar

Goal: Malware Detection
Methodology: Static
Deployment: Off-Device (Portable to On-Device too)

In [83] authors proposed AndroSimilar, an automatic signature generation approach that extracts statistically rare syntactic features for malware detection. Apart from existing malware, AndroSimilar is able to reasonably detect obfuscated malware with techniques like string encryption, method renaming, junk method insertion and changing control flow, widely used to evade fixed anti-malware signature, thus it can detect unknown variants of existing malware. AndroSimilar approach is based on Similarity Digest Hash (SDHash) [119] used in digital forensics to identify similar documents.

Intuitively, completely unrelated apps should have lower probability of having common features. When two unrelated apps share some features, such features should be considered weak as using these shall lead to false positives [120]. Fixed-size byte-sequence features are extracted based on empirical probability of occurrence of their entropy values, then popular features are searched among them according to rarity in neighborhood [119]. Fig. 9 shows the working of AndroSimilar. Following are the steps involved:

- Submit Google Play, third-party or an obfuscated malicious app as input to AndroSimilar.
- Generate entropy values for every byte-sequence of fixed size in a file and normalize these in range of [0, 1000].
- Select statistically robust features according to similarity digest scheme as representative of the app.
- Store extracted features into Bloom Filters. Sequence of Bloom Filters is a signature of an app.
- Compare the signature with the database to detect match with known malware family. If similarity score is beyond a given threshold, mark it as malicious (or repackaged) sample.

Thus, they generate signatures of known malware families as a representative database. If similarity score of an unknown app with any existing family signatures matches beyond a threshold, then it is labeled as malicious. We believe AndroSimilar is a promising approach to detect unseen malware variants.

E. Andrubis

Goal: Malware Analysis and Detection
Methodology: Static and Dynamic
Deployment: Off-Device

Andrubis [121] is a web-based malware analysis platform, built upon some well-known existing tools Droidbox [122], TaintDroid [109], apktool [70] and Androguard [79]. Users can submit suspicious apps through the web based interface. After analyzing the app at the remote-server, Andrubis returns detailed static and dynamic analysis reports as a web page. Andrubis also provides app behavior rating between 0–10, where 0 indicates benign and 10 specifies malicious rating.

To study the Andrubis functionality, a custom SMS based botnet was uploaded on the Andrubis web service. This research prototype rated custom SMS bot with a score 9.9/10. However, none of the commercial anti-malware on the virustotal portal were able to detect this unseen malware. This demonstrates the effectiveness of Andrubis behavior rating against the zero day malware. However, Vidas *et al.* [106] demonstrated that Andrubis virtual environment is detected with anti-analysis techniques and identified the analysis sandbox.

F. APKInspector

Goal: Malware Analysis
Methodology: Static
Deployment: Off-Device

APKInspector [123] is a full-fledged Android static analysis tool, consisting *Ded* [124], *smali/baksmali* [114], *apktool* [70] and *Androguard* [79]. It provides a rich GUI and has following features:

- App meta-data
- Analysis of sensitive permissions

- Displays Dalvik bytecode and Java source code
- Displays control-flow graph
- Displays call-graph, displaying call-in and call-out structures
- Static instrumentation support by allowing modification to the *smali* code

G. Aurasium

Goal: Analysis and Detection
Methodology: Dynamic
Deployment: On-Device

Aurasium [125] is a powerful technique that takes control of execution of apps, by enforcing arbitrary runtime security policies. To be able to do that, Aurasium repackages the Android apps with the policy enforcement module. Aurasium Security Manager component can apply policies on the individual and multiple apps. Any security and privacy violations are reported to the user. Thus, it eliminates the need for manipulating Android OS to monitor app behavior. It intervenes in-case of application accessing sensitive information such as contacts, messages, phone identifiers and executing shell-commands by asking user for confirmation regarding the same.

Aurasium is limited by the fact that it succumbs to the stealth malware, i.e., it can be detected by apps based on signature modification and presence of predefined native library. Malware app may not reveal its malicious behavior if it identifies the presence of Aurasium, hence avoids the detection. As Aurasium depends on repackaging, it may fail to disassemble (or assemble) an code transformed app.

H. Bouncer

Goal: Malware Detection
Methodology: Dynamic
Deployment: Off-Device

Google protects its own app-store, Google Play, with a system called Bouncer. It is a virtual machine based dynamic analysis platform to test the uploaded third party developer apps, before availing them to the users for download. It executes app to look for any malicious behavior and also compares it against previously analyzed malicious apps. Though no documentation of internal functioning is available, Oberheide *et al.* [39] presented their analysis of Bouncer environment by implementing a custom command and control app. Dynamic code loading techniques can evade the Bouncer [126] scrutiny.

I. CopperDroid

Goal: Malware Analysis and Detection
Methodology: Dynamic
Deployment: Off-Device

Reina *et al.* proposed CopperDroid [107], a system which performs system call-centric dynamic analysis of Android apps, using Virtual Machine Introspection. To address the path coverage problem, they have supported the stimulation of events as per the specification present in app manifest file. Authors have shown through experimentation that system call-centric analysis can effectively detect malicious behavior. They have also provided a web interface for other users to analyze apps [127]. However, Vidas *et al.* [106] demonstrate the identification of CopperDroid's virtual environment by employing advanced anti-analysis techniques.

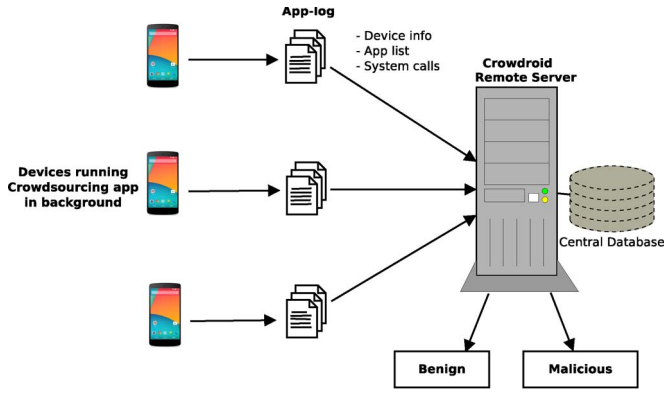


Fig. 10. Crowdroid Architecture.

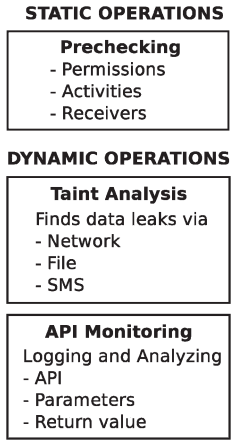


Fig. 11. Features of Droidbox.

J. Crowdroid

Goal: Malware Detection
Methodology: Dynamic
Deployment: Half On-Device, Half Off-Device

Crowdroid [110] is a behavior based malware detection system (see Fig. 10). It has two components, a crowd sourcing app which need to be installed on user-devices and a remote-server for malware detection. The crowd sourcing app sends the behavioral data (i.e., system-call details) in the form of an application log file to the remote server. *Strace*, a system utility present on device is used to collect the system-call details of the apps. The application log file consists of basic device information, list of installed applications and behavioral data. At the remote-server, this data is processed to create feature vectors which could then be analyzed by 2-means partition clustering to predict the app as either benign or malicious. An app report is generated and stored in the database of the remote server.

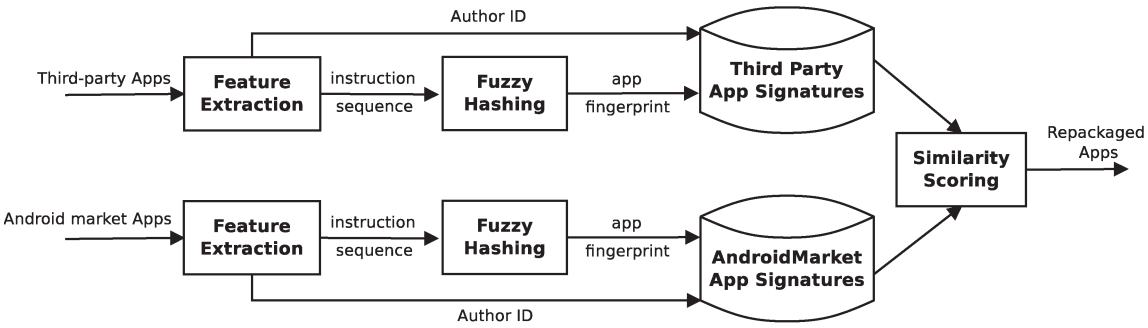


Fig. 12. DroidMOSS Methodology.

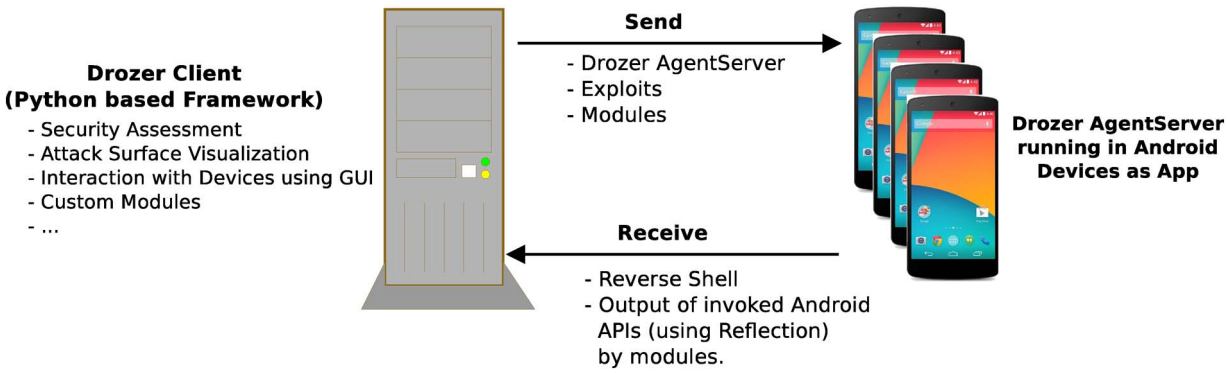


Fig. 13. Working of Drozer.

Results of Crowdroid are accurate for self-written malware and promising for some of the real malware. If the malware is very active, then it is possible to have large difference in system calls, which can help in detection for the same. But, it also suffers with false-positives, as demonstrated by authors using *Monkey Jump2*, an app with *HongTouTou* malware.

Limitation of Crowdroid is, the crowd sourcing app must always be available for monitoring, which can drain the available device resources. Also, this technique is yet to be tested on the known malware families available to ascertain its effectiveness.

K. Droidbox

Goal: Taint Analysis and Monitoring
Methodology: Dynamic
Deployment: Off-Device

Droidbox [122] as illustrated in Fig. 11 is a dynamic analysis tool developed on top of TaintDroid [109]. It modifies the Android framework for API call analysis. Fig. 11 displays the static and dynamic analysis operations of the Droidbox. App analysis begins with the static-pre-checking, which includes parsing permissions, activities and receivers. The app under analysis is executed in emulated environment to perform taint-analysis and API monitoring. Taint-analysis involves labeling (tainting) private and sensitive data that propagates through the program variables, files and interprocess communication.

Taint-analysis keeps track of tainted data that leaves the system either through network, file(s) or SMS and the transmitting app is responsible for ex-filtration. API monitoring involves API logging with its parameters and return values. The results consists the following parameters:

- App hash values
- Network data transferred or received
- File read and write operations
- Data leaks
- Circumvented permissions
- Broadcast receivers
- Services started and classes loaded through `DexClassLoader`
- SMS sent and dialed calls
- Cryptographic operations implemented with Android API
- Temporal operations order
- Tree-map for similarity analysis

Limitation: Droidbox can only monitors the tasks performed within the Android Framework. If the native code leaks the sensitive data, existing system cannot detect and hence the data is ex-filtrated without user knowledge.

L. DroidMOSS

Goal: Repackaged App Detection
Methodology: Static
Deployment: Off-Device

DroidMOSS [4] is an app repackaging detection prototype employing semantic file similarity measures. More specifically, it extracts the DEX opcode sequence of an app and generates a signature fuzzy hashing [128] signature from the opcode.

It also adds developer certificate information, mapped into a unique 32-bit identifier in the signature. Suspected app features are verified against the original apps using the edit-distance algorithm to identify the similarity score. Proposed approach is discussed and illustrated in Fig. 12.

Intuition behind DroidMOSS using the opcodes feature is, it might be easy for adversaries to modify operands, but very hard to change the actual opcodes [4]. This approach has several disadvantages. First, it only considers DEX bytecode, ignoring the native code and app resources. Second, the opcode sequence do not consist high level semantic information and hence generates false negatives. Smart adversary can easily evade this technique using code transformation techniques such as inserting junk bytecode, restructure methods and alter control flow to evade the DroidMOSS prototype.

M. DroidScope

Goal: Analysis
Methodology: Dynamic
Deployment: Off-Device

DroidScope [113] is a Virtual Machine Introspection (VMI) based dynamic analysis Android framework. Unlike other dynamic analysis platforms, it stays out of the emulator and monitors the OS and Dalvik semantics. Hence, even the privilege escalation attacks on the Android kernel can be detected. It also makes the attackers task of disrupting analysis difficult. DroidScope is built upon QEMU emulator with a rich set of APIs to customize the malware analysis prototype. Android malware families DroidKungFu and DroidDream were analyzed and detected with this technique. However, DroidScope's effectiveness against other malware families remains to be tested.

N. Drozer

Goal: Risk Assessment using Exploitation
Methodology: Static and Dynamic
Deployment: Half On-Device, Half Off-Device

Drozer [129] is a comprehensive attack and security assessment framework for Android devices, available as an open-source and a professional version. It allows security enforcement agencies to remotely exploit Android devices to identify vulnerabilities and threats in Android OS. Fig. 13 displays the Drozer functionality. Following is the list of features supported by the Drozer:

- It installs an Agent app on the devices which executes exploitation modules using Java Reflection API. At server-side, one can create their own custom modules in Python and send it to Agent app to perform exploitation activities on the devices.
- It can interact with the Dalvik VM to discover installed packages and related app components. It also allows interaction with the app-components like services, content providers and broadcast receivers to identify vulnerabilities.
- It can create a shell to remotely interact with Android OS.
- It is capable of generating known exploits taking advantage of the already known rooting vulnerabilities.

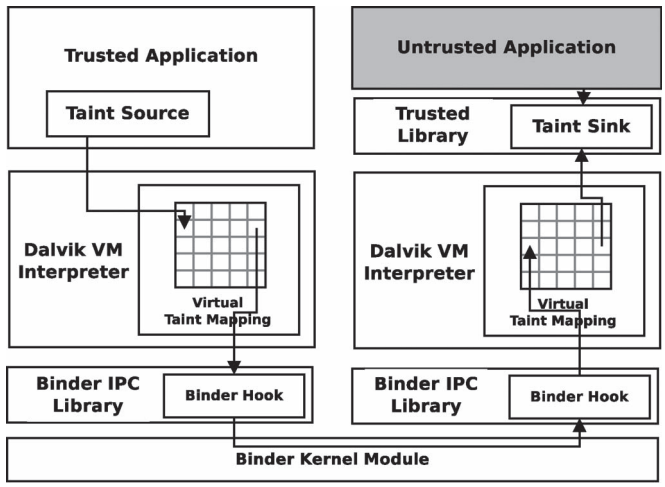


Fig. 14. Taint propagation in TaintDroid.

O. Kirin

Goal: Risk Assessment
Methodology: Static
Deployment: On-Device

In [89] authors proposed a security policy enforcement mechanism, *Kirin*, an on device app vetting framework. Kirin defines a set of rules based on the combination of certain dangerous permissions requested by the app. If an app fails to satisfy the Kirin security rules, the installation is prevented. Thus, the proposed approach decides based on set of rules, on the user behalf.

P. TaintDroid

Goal: Taint Analysis
Methodology: Dynamic and Android Instrumentation
Deployment: Off-Device

TaintDroid [109] extends the Android platform to track the privacy sensitive information leakage in the third-party developer apps. The sensitive data is automatically tainted (or labeled) in order to keep track whether the labeled data leaves the device. When the sensitive data leaves the system, TaintDroid records the label of the particular data and the app which sent the data along with its destination address.

Taint propagation is tracked at four levels of granularity, 1) Variable-level, 2) Method-level, 3) Message-level and 4) File-level. Variable-level tracking uses variable semantics, which provides necessary context to avoid taint propagation. In message-level tracking, the taint on messages is tracked to avoid IPC overhead. Method-level tracking is used for Android native libraries that are not directly accessible to apps but through modified firmware. Lastly file-level tracking ensures integrity of file-access activities by checking whether taint markings are retained.

Let us consider the working of TaintDroid where data of one trusted app is accessed by some untrusted app and sent over the network. The above scenario is displayed in Fig. 14. First, the information of the trusted app is labeled according to its context. A native method interfaces with the Dalvik VM interpreter to store the taint markings in a virtual taint map.

TABLE I
SUMMARY OF ASSESSMENT, ANALYSIS AND DETECTION TOOLS FOR ANDROID PLATFORM ACCORDING TO THEIR GOAL, METHODOLOGY AND DEPLOYMENT. ∗ INDICATES WEB-BASED INTERFACE

Tool	Goal			Methodology						Deployment			Availability
	Assessment	*Analysis*	*Detection*	*Static*	*Dynamic*	*System call/API*	*Profile-based*	*Behavioral*	*VMI*	*On-Device*	*Distributed*	*Off-Device*	
Androguard [79]	✔	✔	✔	✔								✔	Free
Andromaly [81]			✔		✔		✔				✔		Free
AndroSimilar [83]			✔	✔								✔	–
Andrubis [120]		✔	✔	✔	✔		✔	✔				✔	Free
APKInspector [122]		✔		✔								✔	Free
Aurasium [124]			✔		✔			✔				✔	Free*
CopperDroid [126]		✔	✔		✔	✔			✔			✔	Free*
Crowdroid [109]			✔		✔	✔		✔			✔		–
DroidBox [121]		✔			✔	✔		✔				✔	Free
DroidScope [112]		✔			✔	✔		✔	✔			✔	Free
Drozer [128]	✔			✔	✔						✔		Free/Paid
JEB [116]		✔		✔								✔	Paid
Kirin [88]	✔			✔						✔			Free
Paranoid Android [53]			✔		✔			✔				✔	–
TaintDroid [108]			✔		✔	✔		✔				✔	Free

Every interpreter simultaneously propagates the taint tags according to data-flow rules. The Binder library of the TaintDroid is modified to ensure the tainted data of the trusted application is sent as a parcel having a taint tag reflecting the combined taint markings of all contained data. The kernel transfers this parcel transparently to reach the Binder library instance at the untrusted app. The taint tag is retrieved from the parcel and marked to all the contained data by the Binder library instance. Dalvik bytecode interpreter forwards these taint tags along with requested data towards untrusted app component. When that app calls taint sink (for example, network) library, it retrieves taint tag and marks that activity as malicious.

Q. 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 [130] 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 [131] is a promising semantic approach that detects the presence of in-app ad-library with reasonable accuracy compared to existing approaches.

Damopoulos *et al.* [108] proposed a combination of host and cloud based Intrusion Detection System (IDS). In particular, authors highlight the importance of such a system to protect the smartphone when the network resource availability is low, in such case it performs the host based detection. In the device battery is drained, the prototype intelligently opts for the cloud-based detection to leverage the infinite processing and memory. In [132], authors propose an indoor navigation for the visually impaired people in various lighting conditions.

TABLE II
SUMMARY OF WEB BASED ANDROID MALWARE ANALYSIS INTERFACES EMPLOYING DYNAMIC OR A HYBRID ASSESSMENT APPROACH

Property	AASandbox	Andrubis	Apps Playground	CopperDroid	DroidAnalyst	ForeSafe	SmartDroid
Platform Modification	✗	✔	✔	✗	✗	✗	✔
Resource Hogger App Analysis	✗	✗	✗	✔	✔	✗	✗
GUI Interaction	✗	✗	✗	✗	✔	✔	✔
API Hooking	✗	✔	✔	✗	✔	✔	✔
Logcat Analysis	✗	✗	✗	✗	✔	✔	✗
System call Analysis	✔	✗	✗	✗	✔	✔	✔
Risk Prediction with Machine learning model	✔	✔	✗	✔	✔	✗	✗
Anti Anti-Analysis Sandbox	✗	✗	✗	✗	✔	✗	✗
Identifying Data Leakage	✗	✔	✔	✗	✔	✔	✔
Identifying SMS/Call Misuse	✗	✗	✔	✗	✔	✗	✗
Network Traffic Analysis	✗	✔	✗	✗	✔	✔	✗
File Operations Monitoring	✗	✔	✗	✗	✔	✔	✗
Native Code Analysis	✗	✗	✗	✗	✗	✗	✔
On Device Analysis	✗	✔	✗	✗	✗	✔	✗

Proposed prototype PERCEPT-V, a smartphone based UI employs visual tags with a sampling algorithm with different environments, lighting, ambience and usage angles.

Vidas *et al.* [106] proposed a system to identify the emulated Android environment based on differences in behavior, performance evaluation, presence/absence of smartphone hardware and functionality based software capabilities. Such a system highlights the importance of employing *anti* anti-analysis techniques among the sandbox environment. Faruki *et al.* [133] proposed a platform-neutral *anti* anti-emulation sandbox to detect the stealth Android malware. Authors also propose a machine learning model to predict the resource hoggers. Moreover, in [134], 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.* [111] 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 [135]. AsDroid [112] 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.

Portokalidis *et al.* [53] proposed an alternative off-device malware detection approach by cloning smartphone state at remote server. The remote server can have high computing power, more memory and an uninterrupted power supply to execute multiple detection techniques in parallel. The proposed prototype is scalable, practical and incurs a low network overhead. In [136], authors proposed a comparison framework for different dynamic analysis sandbox to identify the limitations among the known web based automatic malware analysis frameworks. Authors concluded that the existing sandbox approaches fail against the advanced and targeted malware. However,

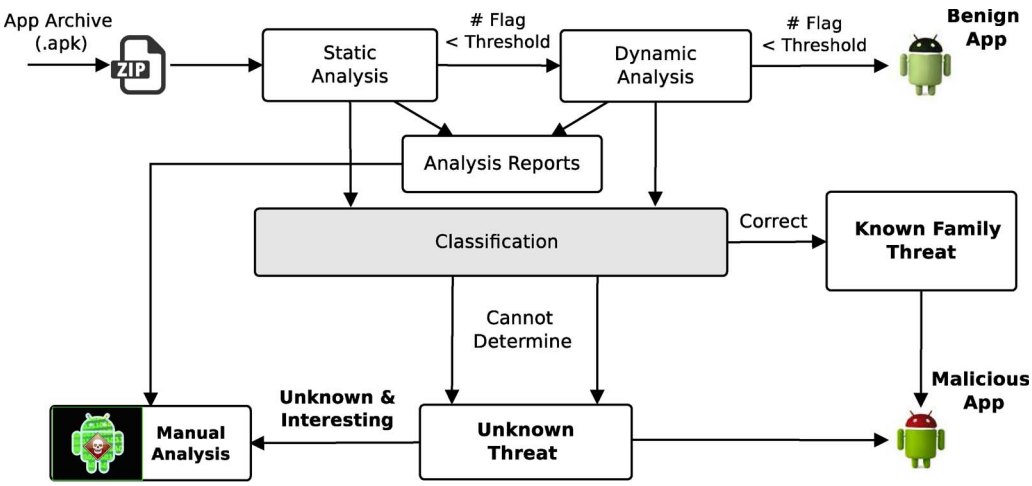


Fig. 15. A Proposed Hybrid Approach for Android Malware Analysis, malware app detection.

Faruki *et al.* [133] proposed a transparent, scalable and Android version invariant sandbox to detect the advanced Android threats.

Rattazi *et al.* [137] proposed a systematic approach that identifies the critical places where access controls are not present or do not properly identify the subject and object of a decision. Authors performed specific experiments to test their hypothesis and concluded that the newer capabilities still need to mature to be used as a sharing environments on mobile Internet devices.

Petsas *et al.* [138] demonstrate advanced malware apps thwarting virtual/emulated environment to hinder dynamic analysis. Authors patched existing malware apps with anti-analysis features to demonstrate the weakness of majority of existing frameworks already discussed in [127], [139]–[143]. However, Faruki *et al.* proposed an Android platform invariant Android Sandbox to uncover advanced malware failing the existing web based analysis sandbox discussed above.

R. Comparison of Web Based Analysis Sandbox

Here, we discuss Sandbox prototypes implemented and available as web service to aid a human analyst keep pace with an exponential increase in Android malware. Andrubis and Copperdroid are implemented on top of Taintdroid and Droidbox, a privacy leakage detection approach and dynamic analysis technique respectively. Table II compares various web based sandbox approaches used by researchers to test and automate malware analysis. Mobile Sandbox is an automated malware analysis and detection approach incorporating native code analysis, a facility not available with the existing prototypes. Droidanalyst is an anti anti-analysis sandbox to detect environment reactive malware. Apps are classified as resource hoggers based on the data transmitted/received. The approaches using Taintdroid or Droidbox have to modify Android version, whereas DroidAnalyst is an android platform neutral sandbox.

IX. Conclusion and Discussion

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 Android 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. In this survey, we discussed Android security architecture and its issues, malware penetration and stealth techniques. In Section VI we discussed *static* and *dynamic* approach for malware analysis and detection. Both approaches can be used separately, but each one has its own limitations. Static analysis can be thwarted by employing encryption and/or transformation techniques discussed in Section V-D. Dynamic analysis can be evaded by several anti-emulation techniques covered in Section VI-B. We also covered prominent malware analysis and detection approaches as summarized in Table I according to their goal, methodology and deployment. Summary shows there is not a single solution that addresses every issue. To tackle wide variety of new malware, a comprehensive evaluation framework incorporating robust static and dynamic methods can be proposed on Android platform.

Manual analysis has become infeasible due to the exponential increase in the number of unknown malware samples. Based on the current reviews, this paper proposes an automated, hybrid approach for Android malware analysis. Architecture of the proposed approach shown in Fig. 15 is our future research direction. As illustrated the APK file is initially dissected with static analysis module. In case of its failure against the encrypted code, dynamic analysis module performs behavioral detection. Static and dynamic analysis will be used to generate app activity reports to enable a malware analyst identify the suspicious sample. Finally, we conclude by highlighting the fact that hybrid detection approaches are gaining prominence in malware analysis.

Appendix

See Listing 1.

```
<uses-permission
  android:name="android.permission.INTERNET" />
<uses-permission
  android:name="android.permission.READ_PHONE_STATE" />
<uses-permission
  android:name="android.permission.RECEIVE_SMS" />

<activity android:label="@string/app_name"
          android:name="com.myapp.Main">
    <intent-filter>
        <action android:name="android.intent.action.
             MAIN" />
        <category
        android:name="android.intent.category.LAUNCHER" /
            >
    </intent-filter>
</activity>

<receiver android:name="com.myapp.SmsReceiver">
    <intent-filter>
        <action
        android:name="android.intent.action.SMS_RECEIVED
            " />
    </intent-filter>
</receiver>

<service android:enabled="true"
         android:name="com.myapp.MyService"
          android:permission="android.permission.INTERNET
              ">
    <intent-filter>
      <action
       android:name="android.intent.action.
           BOOT_COMPLETED" />
    </intent-filter>
</service>

<provider android:name="StudentsProvider"
          android:authorities="com.myapp.MyProvider">
</provider>
```

Listing 1. `AndroidManifest.xml` snippet with declared components.

REFERENCES

[1] G. Inc., Android Smartphone Sales Report, 2013, (Online; Last Accessed Mar. 17, 2014). [Online]. Available: http://www.gartner.com/newsroom/id/2665715

[2] Android Malware Genome Project, (Online; Last Accessed Feb. 11, 2014). [Online]. Available: http://www.malgenomeproject.org/

[3] C. A. Castillo, "Android malware past, present, future," Mobile Working Security Group McAfee, Santa Clara, CA, USA, Tech. Rep., 2012.

[4] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proc. 2nd ACM CODASPY*, New York, NY, USA, 2012, pp. 317–326. [Online]. Available: http://doi.acm.org/10.1145/2133601.2133640

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

[6] Google Bouncer: Protecting the Google Play market, (Online; Last Accessed Oct. 15, 2013). [Online]. Available: http://blog.trendmicro.com/trendlabs-security-intelligence/a-lookat-google-bouncer/

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

[8] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proc. 9th Int. Conf. MobiSys*, New York, NY, USA, 2011, pp. 239–252. [Online]. Available: http://doi.acm.org/10.1145/1999995.2000018

[9] PandaApp, (Online; Last Accessed Mar. 1, 2014). [Online]. Available: http://www.pandaapp.com/

[10] Baidu, (Online; Last Accessed Mar. 1, 2014). [Online]. Available: http://as.baidu.com/

[11] Opera Mobile App Store, (Online; Last Accessed Mar. 1, 2014). [Online]. Available: http://apps.opera.com/en_in/

[12] AppChina, (Online; Last Accessed Mar. 1, 2014). [Online]. Available: http://www.appchina.com/

[13] GetJar, (Online; Last Accessed Mar. 1, 2014). [Online]. Available: http://www.getjar.mobi/

[14] ESET—Trends for 2013, (Online; Last Accessed Feb. 11). [Online]. Available: http://go.eset.com/us/resources/whitepapers/Trends_for_2013_preview.pdf

[15] Kaspersky Security Bulletin 2013, Overall statistics for 2013, (Online; Last Accessed Feb. 11). [Online]. Available: https://www.securelist.com/en/analysis/204792318/Kaspersky_Security_Bulletin_2013_Overall_statistics_for_2013

[16] McAfee Labs Threats Report: Third Quarter 2013, (Online; Last Accessed Feb. 11). [Online]. Available: http://www.mcafee.com/uk/resources/reports/rp-quarterly-threatq3-2013.pdf

[17] F-Secure: Mobile Threat Report Q1 2013, (Online; Last Accessed Feb. 11). [Online]. Available: http://www.fsecure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q1_2013.pdf

[18] F-Secure: Mobile Threat Report Q3 2013, (Online; Last Accessed Feb. 11). [Online]. Available: http://www.fsecure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q3_2013.pdf

[19] F-Secure: Mobile Threat Report H1 2013, (Online; Last Accessed Feb. 11). [Online]. Available: http://www.fsecure.com/static/doc/labs_global/Research/Threat_Report_H1_2013.pdf

[20] VirusTotal, (Online; Last Accessed Feb. 11, 2014). [Online]. Available: https://www.virustotal.com/

[21] Android.Bgserv, (Online; Last Accesed Feb. 12, 2011). [Online]. Available: http://www.symantec.com/security_response/writeup.jsp?docid=2011-031005-2918-99

[22] Backdoor.AndroidOS.Obad.a, (Online; Last Accesed Dec. 25, 2013). [Online]. Available: http://contagiominidump.blogspot.in/2013/06/backdoorandroidosobada.html

[23] RageAgainstTheCage, (Online; Last Accessed Feb. 11). [Online]. Available: https://github.com/bibanon/android-development-codex/blob/master/General/Rooting/rageagainstthecage.md

[24] Android Hipposms, (Online; 2011). [Online]. Available: http://www.csc.ncsu.edu/faculty/jiang/HippoSMS/

[25] Android/NotCompatible Looks Like Piece of PC Botnet, (Online; Last Accesed Dec. 25, 2013). [Online]. Available: http://blogs.mcafee.com/mcafee-labs/androidnotcompatible-looks-like-piece-of-pc-botnet

[26] E. Fernandes, B. Crispo, and M. Conti, "FM 99.9, radio virus: Exploiting FM radio broadcasts for malware deployment," *IEEE Trans. Inf. Forensics Security*, vol. 8, no. 6, pp. 1027–1037, Jun. 2013. [Online]. Available: http://dblp.uni-trier.de/db/journals/tifs/tifs8.html#FernandesCC13

[27] R. Fedler, J. Schütte, and M. Kulicke, "On the Effectiveness of Malware Protection on Android," Fraunhofer AISEC, Berlin, Germany, Tech. Rep., 2013.

[28] C. Jarabek, D. Barrera, and J. Aycock, "ThinAV: Truly lightweight Mobile Cloud-based Anti-malware," in *Proc. 28th Annu. Comput. Security Appl. Conf.*, 2012, pp. 209–218.

[29] Kaspersky Internet Security for Android, (Online; Last Accessed Feb. 11). [Online]. Available: http://www.kaspersky.com/android-security

[30] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "RiskRanker: Scalable and accurate zero-day Android malware detection," in *Proc. 10th Int. Conf. MobiSys*, New York, NY, USA, 2012, pp. 281–294. [Online]. Available: http://doi.acm.org/10.1145/2307636.2307663

[31] G. Suarez-Tangil, J. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Commun. Surveys Tuts.*, vol. 16, no. 2, pp. 961–987, 2014.

[32] M. La Polla, F. Martinelli, and D. Sgandurra, "A survey on security for mobile devices," *IEEE Commun. Surveys Tuts.*, vol. 15, no. 1, pp. 446–471, 2013.

[33] W. Enck, "Defending users against smartphone apps: Techniques and future directions," in *Proc. 7th ICISS*, 2011, pp. 49–70. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-25560-1n_3

[34] Android Security Overview, (Online; Last Accesed Dec. 25, 2013). [Online]. Available: http://source.android.com/devices/tech/security

[35] W. Enck, M. Ongtang, and P. McDaniel, "Understanding android security," *IEEE Security Privacy*, vol. 7, no. 1, pp. 50–57, Jan./Feb. 2009. [Online]. Available: http://dx.doi.org/10.1109/MSP.2009.26

[36] Android Kernel Features, (Online; Last Accessed Mar. 9, 2014). [Online]. Available: http://elinux.org/Android_Kernel_Features

[37] ⟨permission⟩, (Online; Last Accessed Feb. 11). [Online]. Available: http://developer.android.com/guide/topics/manifest/permission-element.html

[38] B. Sanz *et al.*, "PUMA: Permission Usage to detect Malware in Android," in *Proc. Int. Joint Conf. CISIS-ICEUTE-SOCO'Spec. Sessions*, 2013, pp. 289–298.

[39] J. Oberhide, Dissecting the Android Bouncer, (Online; Last Accessed Jun. 1, 2012). [Online]. Available: http://jon.oberheide.org/blog/2012/06/21/dissecting-the-android-bouncer/

[40] Exercising Our Remote Application Removal Feature, (Online; Last Accessed Feb. 11). [Online]. Available: http://androiddevelopers.blogspot.in/2010/06/exercising-our-remote-application.html

[41] CVE, (Online; Last Accessed Feb. 11). [Online]. Available: http://cve.mitre.org/

[42] G. Andre and P. Ramos, "Boxer SMS Trojan," ESET Latin American Lab, Bratislava, Slovakia, Tech. Rep., 2013.

[43] Android Version History, (Online; Last Accessed Mar. 11, 2014). [Online]. Available: http://en.wikipedia.org/wiki/Android_version_history

[44] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, "Upgrading your android, elevating my malware: Privilege escalation through mobile OS updating," in *Proc. IEEE Symp. Security Privacy*, 2014, pp. 393–408.

[45] z4Root, (Online; Last Accessed Feb. 11). [Online]. Available: https://github.com/bibanon/android-developmentcodex/blob/master/General/Rooting/z4root.md

[46] Android Trickery, (Online; Last Accessed Feb. 11). [Online]. Available: http://c-skills.blogspot.com/2010/07/androidtrickery.html

[47] Zimperlich Sources, (Online; Last Accessed Feb. 11). [Online]. Available: http://c-skills.blogspot.in/2011/02/zimperlichsources.html

[48] GingerBreak, (Online; Last Accessed Feb. 11). [Online]. Available: http://forum.xda-developers.com/showthread.php?t=1044765

[49] zergrush, (Online; Last Accessed Feb. 11). [Online]. Available: http://forum.xda-developers.com/showthread.php?t=1296916

[50] Z. Yajin and J. Xuxian, "Dissecting android malware: Characterization and evolution," in *Proc. 33rd IEEE Symp. Security Privacy*, Oakland, CA, USA, 2012, pp. 95–109.

[51] Security Enhancements in Android 4.3, (Online; Last Accesed Dec. 25, 2013). [Online]. Available: http://source.android.com/devices/tech/security/enhancements43.html

[52] Security Enhancements in Android 4.2, (Online; Last Accesed Dec. 25, 2013). [Online]. Available: http://source.android.com/devices/tech/security/enhancements42.html

[53] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: Versatile protection for smartphones," in *Proc. 26th ACSAC*, New York, NY, USA, 2010, pp. 347–356. [Online]. Available: http://doi.acm.org/10.1145/1920261.1920313

[54] Validating Security-Enhanced Linux in Android, (Online; Last Accesed Dec. 25, 2013). [Online]. Available: http://source.android.com/devices/tech/security/se-linux.html

[55] M. Conti, B. Crispo, E. Fernandes, and Y. Zhauniarovich, "CRêPe: A system for enforcing fine-grained context-related policies on android," *Information Forensics and Security, IEEE Trans.*, vol. 7, no. 5, pp. 1426–1438, Oct. 2012.

[56] M. Nauman, S. Khan, and X. Zhang, "Apex: Extending android permission model and enforcement with user-defined runtime constraints," in *Proc. ASIACCS*, D. Feng, D. A. Basin, and P. Liu, Eds., 2010, pp. 328–332. [Online]. Available: http://dblp.uni-trier.de/db/conf/ccs/asiaccs2010.html#NaumanKZ10

[57] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A.-R. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks," Technische Universität Darmstadt, Darmstadt, Germany, Tech. Rep. TR-2011-04, 2011.

[58] M. Ongtang, S. E. McLaughlin, W. Enck, and P. D. McDaniel, "Semantically rich application-centric security in android," in *Proc. ACSAC*, 2009, pp. 340–349. [Online]. Available: http://dblp.uni-trier.de/db/conf/acsac/acsac2009.html#OngtangMEM09

[59] Android Security Analysis Challenge: Tampering Dalvik Bytecode During Runtime, (Online; Last Accessed Feb. 11, 2013). [Online]. Available: http://bluebox.com/labs/android-security-challenge/

[60] "State of mobile security," Lookout Mobile Security, Tech. rep., 2012.

[61] "Current world of mobile threats," Lookout Mobile Security, San Francisco, CA, USA, Tech. rep., 2013.

[62] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, and W. Lee, "The core of the matter: Analyzing malicious traffic in cellular carriers," in *Proc. NDSS*, 2013, vol. 13, pp. 1–16.

[63] H. T. T. Truong *et al.*, "The company you keep: Mobile malware infection rates and inexpensive risk indicators," in *Proc. 23rd Int. Conf. WWW*, 2013, pp. 39–50.

[64] Carat: Collaborative Energy Diagnosis, (Online; Last Accesed Dec. 25, 2013). [Online]. Available: http://carat.cs.berkeley.edu/

[65] Fake Netxflix—Android trojan info stealer, (Online; Last Accessed Feb. 11). [Online]. Available: http://contagiominidump.blogspot.in/2011/10/fake-netxflix-adtroid-trojan-info.html

[66] F. Shahzad, M. A. Akbar, and M. Farooq, "A survey on recent advances in malicious applications analysis and detection techniques for smartphones," National Univ. Comput. Emerging Sci., Islamabad, Pakistan.

[67] Spitmo vs Zitmo: Banking Trojans Target Android, (Online; Last Accessed Feb. 11). [Online]. Available: https://blogs.mcafee.com/mcafee-labs/spitmo-vs-zitmo-banking-trojans-target-android

[68] Fakedefender.B—Android Fake Antivirus, (Online; Last Accesed Dec. 25, 2013). [Online]. Available: http://contagiominidump.blogspot.in/2013/11/fakedefenderb-android-fake-antivirus.html

[69] avast! Free Mobile Security, (Online; Last Accessed Dec. 25, 2013). [Online]. Available: http://www.avast.com/freemobile-security-c?utmexpid=22755838-21.bXJmQHnQA6pakUW6PaLQQ.2&utmreferrer=https%3A%2F%2Fwww.google.com%2F

[70] APKTool, Reverse Engineering with ApkTool, (Online; Accessed Mar. 20, 2013). [Online]. Available: https://code.google.com/android/apk-tool

[71] A. Inc., Class to Dex Conversion with Dx, (Online; Last Accessed Mar. 5, 2013). [Online]. Available: http://developer.android.com/tools/help/index.html

[72] Remote Access Tool Takes Aim with Android APK Binder, (Online; Last Accessed Dec. 25, 2013). [Online]. Available: http://www.symantec.com/connect/blogs/remote-access-tool-takes-aimandroid-apk-binder

[73] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: Evaluating Android anti-malware against transformation attacks," in *Proc. 8th ACM SIGSAC Symp. Inf., Comput. Commun. Security*, 2013, pp. 329–334.

[74] M. Zheng, P. P. C. Lee, and J. C. S. Lui, "ADAM: An automatic and extensible platform to stress test Android anti-virus systems," in *Proc. DIMVA*, 2012, pp. 82–101.

[75] ProGuard, (Online; Last Accessed Feb. 11). [Online]. Available: http://proguard.sourceforge.net/

[76] DexGuard, (Online; Last Accessed Feb. 11). [Online]. Available: http://www.saikoa.com/dexguard

[77] P. Faruki *et al.*, "Evaluation of android anti malware techniques against Dalvik bytecode obfuscation," in *Proc. 13th IEEE Int. Conf. TrustCom*, Beijing, China, Sep. 26–28, 2014.

[78] Dalvik Bytecode Obfuscation on Android, (Online; Last Accessed Feb. 11). [Online]. Available: https://dexlabs.org/blog/bytecode-obfuscation

[79] BlackHat, Reverse Engineering with Androguard, (Online; Accessed Mar. 29, 2013). [Online]. Available: https://code.google.com/androguard

[80] W. Zhou, Y. Zhou, and X. Jiang, "Hey, you get off my market: Detecting malicious apps in official and third party android markets," in *Proc. Annu. NDSS*, New York, NY, USA, 2012, pp. 1–13.

[81] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "'Andromaly': A behavioral malware detection framework for android devices," *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, 2012. [Online]. Available: http://dblp.uni-trier.de/db/journals/jiis/jiis38.html#ShabtaiKEGW12

[82] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: Semantics-based detection of android malware," in *Proc. SIGSOFT FSE*, 2014, pp. 1–12.

[83] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, and A. Bharmal, "AndroSimilar: Robust statistical feature signature for Android malware detection," in *Proc. SIN*, A. Eli, M. S. Gaur, M. A. Orgun, and O. B. Makarevich, Eds., 2013, pp. 152–159. [Online]. Available: http://dblp.uni-trier.de/db/conf/sin/sin2013.html#FarukiGLGB13

[84] A. P. Fuchs, A. Chaudhuri, and J. S. Foster, SCanDroid: Automated security certification of Android applications, Manuscript. [Online]. Available: http://www.cs.umd.edu/~avik/projects/scandroidascaa

[85] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "CHEX: Statically vetting Android apps for component hijacking vulnerabilities," in *Proc. ACM Conf. Comput. Commun. Security*, T. Yu, G. Danezis, and V. D. Gligor, Eds., 2012, pp. 229–240. [Online]. Available: http://dblp.uni-trier.de/db/conf/ccs/ccs2012.html#LuLWLJ12

[86] B. P. Sarma *et al.*, "Android permissions: A perspective combining risks and benefits," in *Proc. 17th ACM Symp. Access Control Models Technol.*, 2012, pp. 13–22.

[87] D. Barrera, H. G. Kayacik, P. C. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *Proc. 17th ACM Conf. CCS*, 2010, pp. 73–84.

[88] C.-Y. Huang, Y.-T. Tsai, and C.-H. Hsu, "Performance evaluation on permission-based detection for android malware," in *Proc. Adv. Intell. Syst. Appl.-Vol. 2*, 2013, vol. 2, pp. 111–120.

[89] W. Enck, M. Ongtang, and P. McDaniel, "On lightweight mobile phone application certification," in *Proc. 16th ACM Conf. Comput. Commun. Security*, 2009, pp. 235–245.

[90] J. Kim, Y. Yoon, K. Yi, J. Shin, and S. Center, "ScanDal: Static analyzer for detecting privacy leaks in Android applications," in *Proc. Workshop MoST*, 2012, in conjunction with the IEEE Symposium on Security and Privacy.

[91] H. S. Karlsen, E. R. Wognsen, M. C. Olesen, and R. R. Hansen, "Study, formalisation, analysis of Dalvik bytecode," in *Proc. 7th Workshop BYTECODE*, 2012, pp. 1–9.

[92] Y. Aafer, W. Du, and H. Yin, "DroidAPIminer: Mining API-level features for robust malware detection in Android," in *Proc. SecureComm*, vol. 127, *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, T. Zia, A. Y. Zomaya, V. Varadharajan, and Z. M. Mao, Eds., Springer, 2013, pp. 86–103. [Online]. Available: http://dblp.uni-trier.de/db/conf/securecomm/securecomm2013.html#AaferDY13

[93] M. Zheng, M. Sun, and J. C. S. Lui, "DroidAnalytics: A signature based analytic system to collect, extract, analyze and associate android malware," in *Proc. 12th IEEE Int. Conf. TrustCom*, 2013, pp. 163–171.

[94] JD-GUI, Android Decompiling with JD-GUI, (Online; Last Accessed Mar. 1, 2014). [Online]. Available: http://java.decompiler.free.fr/?q=jdgui

[95] JAD, JAD Java Decompiler, (Online; Last Accessed Mar. 1, 2014). [Online]. Available: http://varaneckas.com/jad/

[96] H. van Vliet, Mocha, The Java Decompiler, (Online; Last Accessed Mar. 1, 2014). [Online]. Available: http://www.brouhaha.com/_eric/software/mocha/

[97] SOOT, Soot: A Java optimization framework, (Online; Accessed Mar. 1, 2014). [Online]. Available: http://www.sable.mcgill.ca/soot/

[98] WALA, T. J. Watson Libraries for Analysis (WALA), (Online; Accessed Mar. 1, 2014). [Online]. Available: http://wala.sourceforge.net/wiki/index.php/

[99] H. Inc., Fortify static code analyzer, (Online; Accessed Mar. 1, 2014). [Online]. Available: http://www8.hp.com/us/en/softwaresolutions/software.html?compURI=1338812

[100] E. William, O. Damien, M. Patrick, and C. Swarat, "A study of Android application security," in *Proc. USENIX*, San Francisco, CA, USA, 2011, p. 163.

[101] D. Octeau, S. Jha, and P. McDaniel, "Retargeting Android applications to Java bytecode," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, p. 6.

[102] A. Bartel, J. Klein, Y. Le Traon, and M. Monperrus, "Dexpler: Converting Android Dalvik bytecode to Jimple for static analysis with Soot," in *Proc. ACM SIGPLAN Int. Workshop State Art Java Program Anal.*, 2012, pp. 27–38.

[103] C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale," in *Proc. Trust Trustworthy Comput.*, 2012, pp. 291–307.

[104] Dex2Jar, Android Decompiling with Dex2jar, (Online; Last Accessed May 15, 2013). [Online]. Available: http://code.google.com/p/dex2jar/

[105] UI/Application Exercise Monkey, (Online; Last Accessed Feb. 11). [Online]. Available: http://developer.android.com/tools/help/monkey.html

[106] T. Vidas and N. Christin, "Evading android runtime analysis via sandbox detection," in *Proc. 9th ACM ASIA CCS*, New York, NY, USA, 2014, pp. 447–458. [Online]. Available: http://doi.acm.org/10.1145/2590296.2590325

[107] A. Reina, A. Fattori, and L. Cavallaro, "A system call-centric analysis and stimulation technique to automatically reconstruct Android malware behaviors," in *Proc. EUROSEC*, Prague, Czech Republic.

[108] D. Damopoulos, G. Kambourakis, and G. Portokalidis, "The best of both worlds: A framework for the synergistic operation of host and cloud anomaly-based IDS for smartphones," in *Proc. 7th EuroSec*, New York, NY, USA, 2014, pp. 6:1–6:6. [Online]. Available: http://doi.acm.org/10.1145/2592791.2592797

[109] E. William, G. Peter, C. Byunggon, and C. Landon, "TaintDroid: An information flow tracking system for realtime privacy monitoring on smartphones," in *Proc. USENIX*, 2011.

[110] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-based malware detection system for Android," in *Proc. 1st ACM Workshop Security Privacy Smartphones Mobile Devices*, 2011, pp. 15–26.

[111] K. O. Elish, D. (Daphne) Yao, and B. G. Ryder, "User-centric dependence analysis for identifying malicious mobile apps," in *Proc. Workshop MoST*, 2012.

[112] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, "AsDroid: Detecting stealthy behaviors in android applications by user interface and program behavior contradiction," in *Proc. ICSE*, 2014, pp. 1036–1046.

[113] L. K. Yan and H. Yin, "DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis," in *Proc. 21st USENIX Security Symp.*, 2012, p. 29.

[114] BakSmali, Reverse Engineering with Smali/Baksmali, (Online; Accessed Mar. 20, 2013). [Online]. Available: https://code.google.com/smali

[115] DARE: Dalvik Retargeting, (Online; Last Accessed Feb. 11, 2013). [Online]. Available: http://siis.cse.psu.edu/dare/

[116] Dedexer, (Online; Last Accessed Feb. 11, 2013). [Online]. http://dedexer.sourceforge.net/

[117] JEB Decompiler, (Online; Last Accessed Feb. 11, 2013). [Online]. Available: http://www.android-decompiler.com/

[118] Similarities for Fun & Profit.

[119] V. Roussev, "Data fingerprinting with similarity hashes, advances in digital forensics," in *Proc. Int. Conf. Digit. Forensics*, 2010, pp. 207–226.

[120] V. Roussev, "Building a better similarity trap with statistically improbable features," in *Proc. 42nd HICSS*, 2009, pp. 1–10.

[121] Andrubis, 2012. [Online]. Available: http://anubis.iseclab.org/

[122] A. Desnos and P. Lantz, "Droidbox: An android application sandbox for dynamic analysis, 2011. [Online]. Available: https://code.google.com/p/droidbox/

[123] APKInspector, 2013. [Online]. Available: https://github.com/honeynet/apkinspector/

[124] ded: Decompiling Android Applications, (Online; Last Accessed Feb. 11). [Online]. Available: http://siis.cse.psu.edu/ded/

[125] R. Xu, H. Saïdi, and R. Anderson, "Aurasium: Practical policy enforcement for Android applications," in *Proc. 21st USENIX Conf. Security Symp.*, 2012, pp. 27–27, USENIX Association.

[126] Google Bouncer: Bad guys may have an app for that, Feb. 2012. [Online]. Available: http://www.techrepublic.com/blog/it-security/google-bouncer-badguys-may-have-an-app-for-that/7422/

[127] CopperDroid, Feb. 2012. [Online]. Available: http://copperdroid.isg.rhul.ac.uk/copperdroid/index.php

[128] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digit. Investigation*, vol. 3, pp. 91–97, Sep. 2006. [Online]. Available: http://dx.doi.org/10.1016/j.diin.2006.06.015

[129] Drozer—A Comprehensive Security and Attack Framework for Android, (Online; Last Accessed Feb. 11, 2013). [Online]. Available: https://www.mwrinfosecurity.com/products/drozer/

[130] M. C. Grace, W. Zhou, X. Jiang, and A.-R. Sadeghi, "Unsafe exposure analysis of mobile in-app advertisements," in *Proc. 5th ACM Conf. WISEC*, New York, NY, USA, 2012, pp. 101–112. [Online]. Available: http://doi.acm.org/10.1145/2185448.2185464

[131] A. Narayanan, L. Chen, and C. K. Chan, "Addetect: Automated detection of android ad libraries using semantic analysis," in *Proc. IEEE 9th Int. Conf.ISSNIP*, Singapore, Apr. 21–24, 2014, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/ISSNIP.2014.6827639

[132] H. Dong, J. Kang, J. Schafer, and A. Ganz, "Android-based visual tag detection for visually impaired users: System design and testing," *Int. J. E-Health Med. Commun.*, vol. 5, no. 1, pp. 63–80, 2014. [Online]. Available: http://dx.doi.org/10.4018/ijehmc.2014010104

[133] P. Faruki, V. Ganmoor, L. Vijay, M. Gaur, and M. Conti, "Android Platform Invariant Sandbox for Analyzing Malware and Resource Hogger apps," in *Proc. 10th IEEE Int. Conf. SecureComm*, Beijing, China, Sep. 26–28, 2014, pp. 1–6.

[134] G. Suarez-Tangil, M. Conti, J. E. Tapiador, and P. Peris-Lopez, "Detecting targeted smartphone malware with behavior-triggering stochastic models," in *Proc. Eur. Symp. Res. Comput. Security*, 2014, pp. 183–201.

[135] Dendroid malware can take over your camera, record audio, sneak into Google Play, (Online; 2014). [Online]. Available: https://blog.lookout.com/blog/2014/03/06/dendroid/

[136] S. Neuner *et al.*, "Enter sandbox: Android sandbox comparison," in *Proc. IEEE MoST*, 2014.

[137] P. Ratazzi *et al.*, "A systematic security evaluation of android's multi-user framework," in *Proc. IEEE MoST*, 2014, pp. 1–10.

[138] T. Petsas, G. Voyatzis, E. Athanasopoulos, M. Polychronakis, and S. Ioannidis, "Rage against the virtual machine: Hindering dynamic analysis of android malware," in *Proc. 7th Eur. Workshop Syst. Security*, 2014, Art. ID. 5.

[139] M. Lindorfer, Andrubis: A tool for analyzing unknown android applications. [Online]. Available: http://www.seclab.tuwien.ac.at/papers/andrubis_badgers14.pdf

[140] M. Lindorfer *et al.*, "Andrubis—1,000,000 apps later: A view on current Android malware behaviors," in *Proc. 3rd Int. Workshop BADGERS*, 2014, pp. 1–15.

[141] L. Weichselbaum *et al.*, "Andrubis: Android malware under the magnifying glass," Vienna University of Technology, Wien, Austria, Tech. Rep. TR-ISECLAB-0414-001, 2014.
[142] T. Bläsing, L. Batyuk, A.-D. Schmidt, S. A. Çamtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *Proc. MALWARE*, 2010, pp. 55–62.
[143] C. Zheng *et al.*, "SmartDroid: An automatic system for revealing UI-based trigger conditions in android applications," in *Proc. 2nd ACM Workshop SPSM*, New York, NY, USA, 2012, pp. 93–104. [Online]. Available: http://doi.acm.org/10.1145/2381934.2381950

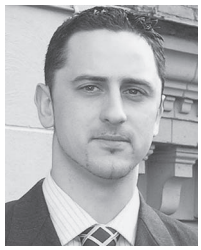
Parvez Faruki received the bachelor's degree in computer engineering from L.D. Engineering, Ahmedabad, India, in 2000 and the M.Tech degree from the Malaviya National Institute of Technology (MNIT), Jaipur, India, in 2012, where he is pursuing doctoral research on android security with the CSE Department. He has conceptualized DroidAnalyst and proposed AbNORMAL and AndroSimilar analysis techniques. His research interests include malware analysis, mobile platform security, and machine learning techniques.

Ammar Bharmal is an M.Tech Scholar working on Android malware analysis with the CSE Department, Malaviya National Institute of Technology, Jaipur, India. He has been actively involved in mobile platform security research since 2012.

Vijay Laxmi received the BTech degree in ECE from JNV University, Rajasthan, India, in 1991, the MTech degree in CSE from IIT Delhi, New Delhi, India, in 1992, and the Ph.D. degree in ECS in 2003 from the University of Southampton, Southampton, U.K., under Commonwealth Scholarship and Fellowship. She is an Associate Professor in computer science and engineering at the Malaviya National Institute of Technology, Jaipur, India. As a Principal Investigator, she has completed three research projects. She is actively involved in malware research. She has supervised seven Ph.D. candidates, four covering different aspects of information security. To date, she has more than 60 papers in refereed conferences/journals in the area of information security.

Vijay Ganmoor is an M.Tech Scholar working on Android malware analysis with the CSE Department, Malaviya National Institute of Technology, Jaipur, India. He has been actively involved in mobile platform security research since 2012.

Manoj Singh Gaur received the BE(ECE) degree from JNV University, Rajasthan, India, in 1988, the ME(CSE) degree from IISc Bangalore, Bangalore, India in 1994, and the Ph.D. degree in ECS from the University of Southampton, Southampton, U.K., in 2004. He has been teaching for the past 25 years. He is a Professor in computer science and engineering at the Malaviya National Institute of Technology, Jaipur, India. He has supervised seven Ph.D. candidates, four covering different aspects of information security. He has also successfully completed six research projects. His research interests include information security, particularly malware analysis.



Mauro Conti (SM'10) received the Ph.D. degree from the Sapienza University of Rome, Rome, Italy, in 2009. After his Ph.D., he was a Postdoctoral Researcher at Vrije Universiteit Amsterdam, Amsterdam, The Netherlands. He was a Visiting Researcher at GMU (2008), UCLA (2010), UCI (2012–2014), and TU Darmstadt (2013). He is an Associate Professor at the University of Padua, Padua, Italy. His main research interest is in the area of security and privacy. In this area, he published more than 80 papers in topmost international peer-reviewed journals and conferences. Dr. Conti was a recipient of a Marie Curie Fellowship (2012) from the European Commission and a Fellowship from the German DAAD (2013). He served as a Program Committee Member of several conferences. He served as a Panelist at the ACM CODASPY 2011 and the General Chair for SecureComm 2012 and ACM SACMAT 2013.

Muttukrishnan Rajarajan received the Ph.D. degree from the City University London, London, U.K., in 2001. He is a Professor in security engineering at the City University London. His research expertise is in the areas of mobile security, intrusion detection, and privacy techniques. He has chaired several international conferences in the area of information security and involved in the editorial boards of several security and network journals. He is also a Visiting Research Fellow at the British Telecommunications UK and is currently actively engaged in the U.K. Government's Identity Assurance Program. He is a Member of the ACM and an Advisory Board Member of the Institute of Information Security Professionals UK.