City Research Online

City, University of London Institutional Repository

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

Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur and Mauro Conti

Abstract—**Android smartphones are gaining big market share due to several reasons, including open architecture and popularity of its application programming interfaces (APIs) in developer community. In general, smartphone has become pervasive due to its cost effectiveness, ease of use and availability of office applications, Internet, games, vehicle guidance using location-based services apart from conventional voice calls, messaging and multimedia services.**

Increase in number of Android smartphone and associated monetary benefits has led to an exponential rise in Android malware apps between 2011-2014. Academic researchers and commercial anti-malware companies have realized that conventional signature based and static analysis methods are vulnerable against prevalent stealth techniques such as encryption, code transformation and analysis environment detection approach. This realization has led to the use of behavior based, anomaly based and dynamic analysis methods. As one single approach may be ineffective against above techniques, complementary approaches may be combined for effective malware app detection.

Though many reviews extensively cover smartphone OS security, as Android smarthphone have captured more than 75% market, we believe a deep examination of Android security, malware growth, anti analysis methods and mitigation solution specifically for android is required. In this review, we discuss Android security enforcement and its issues, Android malware growth timeline between 2010-2013, malware penetration and anti-analysis techniques used by malware authors to bypass analysis methods. This review gives an insight into the strength and weakness of known research methodologies and thus provide a platform for research practitioners towards proposing next generation Android security, malware analysis and malicious app detection methods.

Index Terms—**Android Malware, Static Analysis, Dynamic Analysis, Behavioral Analysis, Obfuscation, Stealth Malware**

I. INTRODUCTION

Android smartphone Operating System has captured more than 75% of the total market-share, leaving its competitors iOS, Windows Phone and Blackberry far behind [1]. Even though smartphones were used in the previous decade, launch of iOS and Android has changed the landscape by generating an enormous attraction worldwide among consumers and developers alike. Smartphones have become ubiquitous due to wide range of connectivity options, such as GSM, CDMA, Wi-Fi, GPS, Bluetooth and NFC. Gartner report of year 2013 shows an increase of 42.3% in smartphone sales from 2012 [1]. Comparison between total sales in year 2012 and 2013 is

P. Faruki, A. Bharmal, V. Laxmi, V. Ganmoor and Manoj Singh Gaur are with Computer Engineering Department, MNIT Jaipur, India (e-mail: parvez@mnit.ac.in; vlaxmi@mnit.ac.in; gaurms@gmail.com)

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

shown in the Figure 1. It shows an increase of 12% from 66 to 78 for Android and its nearest competitor iOS's sale declines by 4% from 19 to 15. Always-on internet connectivity and personal information such as contacts, messages, social network access, browsing history, bank transactions have attracted malware developers also towards smartphone OS platforms in general and Android in particular due to its popularity. This has led to the rise of Android malware such as premium-rate SMS Trojan, spyware, botnets, aggressive adware and privilege escalation exploits distributed through third-party and official app-stores.
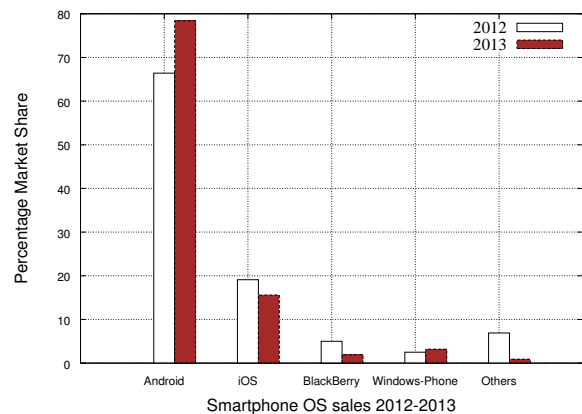

Fig. 1: Mobile OS sales comparison between 2012 and 2013 [1]

Android's popularity among users has made the developers provide innovative applications (popularly called *apps*). Google Play, official Android app market hosts third-party developer apps with a nominal fee providing moderate control. Google Play hosts more than one million apps [2] with large number of downloads each day. Unlike Apple market app-store, Google Play does not verify uploaded apps manually. Instead, Google Play relies on Bouncer, a dynamic emulation environment to protect itself from malicious app threats. It would provide protection against threats, but cannot analyze the vulnerability of existing apps [3]. Malicious apps may trick vulnerable apps to divulge user's private information that inadvertently harms the reputation of the latter. Moreover, Android does not recommend, but allows installation of third-party apps on device, which has stirred up dozens of regional as well as international app-stores [4] [5] [6] [7] [8]. However, protection and quality of apps available in third-party app-stores is a matter of concern [9].

Android security solution providers report an alarming rise

of malware from just three families and mere 100 samples in 2010, to more than hundred families with 0.12-0.6 million unique samples [10] [11] [12] [13] [14] [15]. The number of malicious apps uploaded on VirusTotal [16] is doubling every year. Malicious apps are using clever ways to bypass existing security mechanisms provided by Android OS as well as anti-malware products such as stealth techniques, dynamic execution, code obfuscation, repackaging and encryption [17] [18]. Existing malware propagate by employing above techniques to defeat signature-based approach used by anti-malware products. Thus, new mechanisms that adapt and provide timely response to such techniques are important. Proactive approaches are needed to detect unknown variants of known malware with less number of signature updates, in-contrast to one signature for each known malware.

Malware app developers gain smartphone control by exploiting platform vulnerabilities [19], stealing sensitive user information [17], getting monetary benefits by exploiting telephony services [20] or creating botnet [21]. Thus, it is important to understand their operational activities, mode of working and usage pattern in recent past to devise proactive detection methods.

Huge increase in malicious apps has forced anti-malware industry to carve out robust methods for efficient detection on device under existing constraints. Majority of anti-malware still employ retrospective signature based detection due to implementation simplicity and efficiency [22]. Signature based methods can be easily circumvented through code obfuscation, necessitating a new signature for every malicious sample [23] and that is why an anti-malware client has to regularly update its signature database. Due to limited processing capability and constrained battery power on a smartphone, cloud-based solutions for analysis and detection came into existence [24] [25]. Signature generation needs expertise and patience for each malware sample as it may incur false positives while detecting unknown variants of a known malware family. Due to increasing number of malware and their variants, there is a need to employ automatic signature generation and detection incurring low false positive rate.

Off-device detailed analysis of malware is required to understand its functionality. Analysis of samples can be done manually to extract robust signatures out of them. However, given the rapid rise of malware, there is need for analysis methods that need minimum human intervention, helping malware analyst to generate timely solution of new malware. Static analysis can quickly and precisely identify malicious patterns, but fails against code obfuscation as well as dynamic code execution on Android [26]. Thus, dynamic analysis approaches, though time-consuming, are used to extract malicious behavior of samples using stealth techniques, by executing them in a sandbox environment.

Academic and industry researchers have proposed many solutions and frameworks to mitigate malicious app threats since the launch of Android in 2008, some of which are open-source. These solutions can be characterized basically using following three parameters:

1) *Goal* of the proposed solution can be either app-security assessment, analysis or malware detection. App-security assessment solutions try to find out vulnerabilities in apps, which if exploited by an adversary, can harm the user and device security. Analysis solutions check for malicious behavior within unknown apps, whereas detection solutions aim to prevent existing malware from installing on the device.

2) *Methodology* to achieve above goals can be *static* analysis based approach that is used to identify behavior of apps without actually executing them. Control-flow and data-flow analysis are example implementations of formal static analysis. In *Dynamic* analysis based approach, apps are executed/emulated in a sandboxed environment, in order to monitor their activities and identify behaviors, which are otherwise difficult or impossible using static analysis approach.

3) *Deployment* of the above solutions.

Existing survey papers on smartphone security review the state of the art in general considering all popular OS platforms [27] [28], while this review paper mainly focuses on Android OS. In particular, La Polla et al. [28] surveyed smartphone security threats and their solutions for the period 2004-2011, which has very limited coverage for Android OS.

Suarez-Tangil et al. [27] extended the work of La Polla et al. [28]. In particular, they concentrated on attacks based on related smartphone feature misuse such as hardware, communication, sensors and system, which gives good insight into how utilizing certain features of Android will affect overall security of the device. Suarez-Tangil et al. categorized malware based on their attack goals, distribution & infection and privilege acquisition. On the contrary, we categorize malware as per anti-malware industry's terminology, which aims to provide more accurate view of malware infection rate and threat perception for period 2010-13.

In 2011, William Enck [29] studied the security mechanisms available in Android, particularly, protection through permissions and security implications of inter-app communication. Moreover, he discussed other third-party Android platform hardening solutions, their benefits and limitations. He also examined various app security analysis proposals and gave future direction to enhance them.

We aim to complement former reviews by expanding the coverage of Android security issues, malware growth during 2010-13, their penetration, stealth techniques and strength as well as weaknesses of some of the popular mitigation solutions. In particular, we comprehensively cover stealth techniques used by malware authors to evade detection by generating variants of existing Android malware. We also propose a hybrid framework for Android malware analysis and detection, which gives insight into our future research direction. This survey paper is organized as follows:

- Section II discusses the Android architecture, application structure and inter–component communication.
- Section III discusses security enforcement done at various level within Android and Section IV covers its issues with respect to user's security.
- Section V categorizes Android malware according to their functionality and Section VI covers various penetration techniques used by them.

- Section VII discusses obfuscation and stealth techniques employed by malware.
- Section IX outlines assessment, analysis and detection methodology and their deployment methods.
- Section X reviews state of the art tools for app-security assessment, analysis and malware detection proposed by academia and anti-malware industry along with their strength and drawbacks.
- Finally, Section XI concludes the survey by evaluating state of the art tools. We also propose an Android malware analysis and detection framework that employs both static and dynamic analysis techniques, as a recommendation for future research direction.

II. BACKGROUND

Android is being developed under Android Open Source Project (AOSP), maintained by Google and promoted by Open Handset Alliance (OHA), which consists Original Equipment Manufacturers (OEMs), chip-makers, carriers and developers. Android apps are developed in Java, however, native code and shared libraries are coded in C/C++. Typical Android architecture is shown in Figure 2. The bottom layer is Linux kernel tuned specifically for embedded environment with limited resources. Android is based on Linux kernel due to its robust driver model, existing drivers, memory and process management, networking support along with other core services. Currently, Android fully supports two Instruction Set Architectures: 1) ARM, prevalent in smartphones, Tablets; 2) x86, prevalent among Mobile Internet Devices (MIDs). On the top of Linux kernel are native libraries to support high performance third-party reusable libraries and in-house functionality. Java code is translated into Dalvik byte code that runs under newly created Java runtime, called Dalvik Virtual Machine, optimized for limited resource availability on mobile platform. After booting of OS completes, a process called *zygote* initializes Dalvik VM by pre-loading all core libraries and waits, through a socket, for new process creation requests to be forked from itself. This makes new app-process creation very fast. Finally, application framework layer provides uniform and concise view of Java libraries for use by apps. Android runs its sensitive functionality such as telephony, GPS, network, power-management, radio and media as system services, which are again protected with permissions.

A. App Structure

An Android app is packaged into a .apk file, which is technically a zip archive, consisting of several files and folders as shown in Figure 3. In particular, AndroidManifest.xml file contains meta-data about an app, such as package name, permissions required, definition of one or more components like Activities, Services, Broadcast Receivers or Content Providers, minimum and maximum platform version supported, libraries to be linked against and so forth. res folder consist of icons, images, string, numeric, color constants, UI layout, menus, animations etc. compiled into binary format. assets folder contains non-compiled resources and its directory structure is maintained. classes.dex contains Dalvik executable bytecode to be run under Dalvik Virtual Machine. META-INF folder contains app digital signature, as well as, developer certificate used for verification and identification respectively.

As mentioned before, Android apps are written in Java language. App building process is shown in the Figure 4. Compilation of Java code creates number of .class files, containing intermediate Java-bytecode, for each Java class in source. Using *dx* tool, those .class files are converted into a single *Dalvik Executable (dex)* file. Dex file contains Dalvik
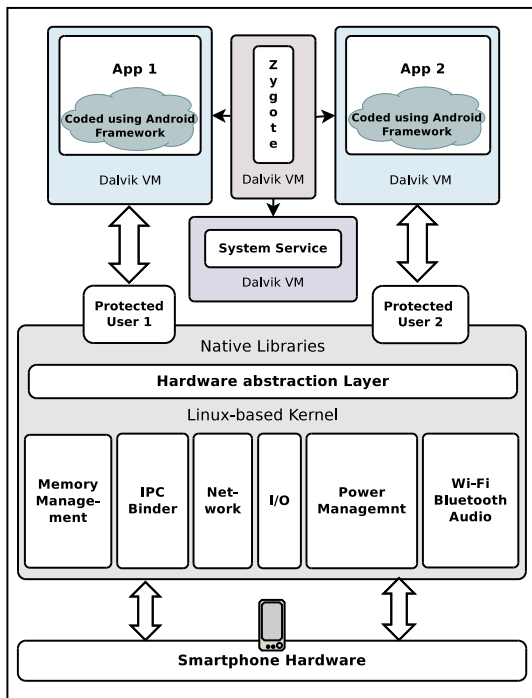

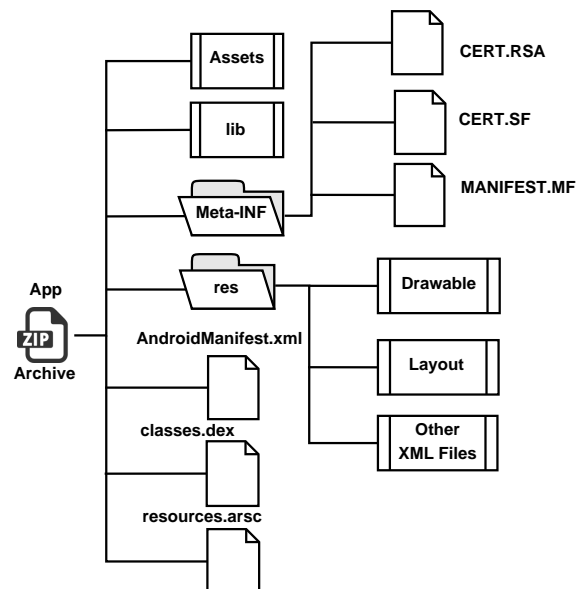Fig. 2: Android Architecture [30]

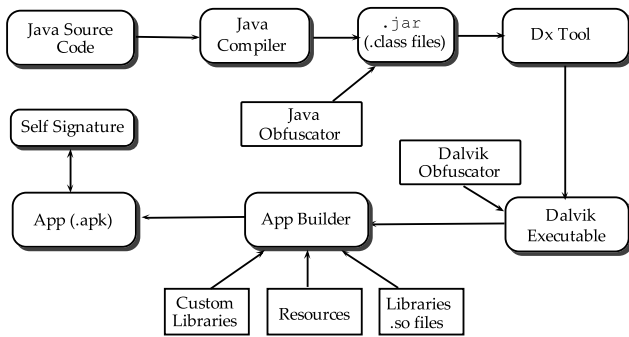Fig. 3: Android PacKage (APK) Structure

Fig. 4: App Building Process

bytecode, which runs under register-based Dalvik Virtual Machine, unlike stack-based JVM.

B. App Components

In Android, an app is functionally divided into one or more components given below:

- *Activity*: It is the user interface component of an app. Arbitrary number of activities can be declared in manifest file depending on the requirements. Apart from performing some pre-defined task, an activity can also return the result to its caller. Activities are launched using *Intents* (explained in the next subsection).
- *Service*: It is a component without any user interface. A service is generally used to perform background processing, for example, playing an audio or downloading data from network. Services are launched using *Intents* (explained in the next subsection).
- *Broadcast Receiver*: This component listens to the events, which are generated by the system, for example, BOOT_COMPLETED, SMS_RECEIVED etc. Other apps can also broadcast their application-defined events, which can be handled by other apps using this component.
- *Content Provider*: This component is also known as *data-store*, providing a consistent interface for data access within app or to other apps. Externally, data within the content provider appears in the form of relational database, but internally it can have completely different storage implementation. Data-store is accessible through application-defined Uniform Resource Identifiers (URIs).

Each component may be accessible to other apps, depending upon whether they are exported. Listing 1 shows an example definition for each of the component in `AndroidManifest.xml`. Each component can get invoked or executed independent of others. Thus, Android app has multiple entry-points, depending upon the number of components it is made-up of.

C. Inter-component Communication

Exported components of apps interact with each other using a high-level abstraction for inter-process communication, called *Intent*, internally handled by Binder driver. Apps invoke *activities* and *services* as well as send broadcast events using Intents only. Also, system events are broadcasted through

Intents. Intent can contain explicit address of the receiver component using class/package name field. Otherwise, depending upon the presence of action, category and data fields, system implicitly sends Intent to the matching one or more receiver component. Each component registers itself to receive Intent(s) using one or more *intent-filter*, which also specifies the kind of action, category and/or data it can accept. For example, in Listing 1, *service* component will be invoked only when it receives system Intent with action equals to BOOT_COMPLETED.

```xml
1  <uses-permission
2    android:name="android.permission.INTERNET"/>
3  <uses-permission
4    android:name="android.permission.READ_PHONE_STATE"/>
5  <uses-permission
6    android:name="android.permission.RECEIVE_SMS"/>
7
8  <activity android:label="@string/app_name"
9            android:name="com.myapp.Main">
10     <intent-filter>
11         <action android:name="android.intent.action.
                MAIN"/>
12         <category
13         android:name="android.intent.category.LAUNCHER"/
                >
14     </intent-filter>
15 </activity>
16
17 <receiver android:name="com.myapp.SmsReceiver">
18     <intent-filter>
19         <action
20         android:name="android.intent.action.SMS_RECEIVED
                "/>
21     </intent-filter>
22 </receiver>
23
24 <service android:enabled="true"
25         android:name="com.myapp.MyService"
26         android:permission="android.permission.INTERNET
                ">
27     <intent-filter>
28       <action
29       android:name="android.intent.action.
                BOOT_COMPLETED"/>
30     </intent-filter>
31 </service>
32
33 <provider android:name="StudentsProvider"
34           android:authorities="com.myapp.MyProvider">
35 </provider>
```

Listing 1: Snippet from `AndroidManifest.xml` with components

III. ANDROID SECURITY ENFORCEMENT

Android has been designed with security in mind from the very inception with the aim to protect user data, apps, the device and the network [30]. However, overall security depends on the developers' willingness and capability to employ best practices. Also, user must be aware of the effect that some app can have after installation, on its data and device's security. Anti-malware solutions on Android cannot handle malware aggressively due to security model enforced on apps. For example, anti-malware apps have limited scanning and/or monitoring capability for other apps or file-system in the device. In this section, we revise the security features provided by Android platform.

A. Application Sandboxing

At kernel level, Android utilizes DAC (Discretionary Access Control) feature of Linux, by assigning every app process a

unique UID, so that an app cannot interfere with other apps or system services. Android also protects network access by implementing a feature called **Paranoid Network Security**, through which Wi-Fi, Bluetooth, Internet access services run in different groups [31]. If some app has been granted permission for particular network access (e.g., Bluetooth), process of that app is assigned into corresponding group. Thus, apart from UIDs, a process may get assigned one or more GIDs. Android app sandboxing is shown in Figure 5.

An app must contain a PKI certificate signed by the developer who created it (see Figure 4). Signing is the point of trust between Google and developers, so that developers are sure that their apps are provided to the users unmodified, and they only are responsible for their apps' behavior. This signing is used for placing an app to its sandbox by assigning unique UID. If certificate of an app A matches with some already installed app B on the device, Android assigns the same UID (i.e., sandbox) to app A as app B, allowing them to share each others' private files and permissions. For this reason, it is strictly not advisable that developer should share its own certificate with others.

B. Permissions at Framework-level

To restrict every app from accessing important functionality of a smartphone such as telephony, network, contacts/SMS/sdcard and GPS location, Android provides permission-based security model at application framework level. App must declare the permissions it want to access using `<uses-permissions>` tag in `AndroidManifest.xml` file as shown in the Listing 1. By default, app has no permission to perform actions that would affect other apps, system or user, and thus, it runs with a very limited capability. Thus, restrictions are enforced, on specific operations an app (process) can perform, at the time of installation itself.

Android permissions are divided into following four protection-levels [32]:

1) *Normal*: These permissions has minimal risk for the device, system and users. They are granted automatically at the time of installation.
2) *Dangerous*: These permissions has higher risk and give app the access of private data and important features of the device. They must be granted by users before installation.
3) *Signature*: These permissions are granted only if requesting app is signed with the same certificate as the app that declared the permissions. They are granted automatically at the time of installation.
4) *SignatureOrSystem*: These permissions are granted only if requesting app is signed with the same certificate as the Android system image or app that declared the permissions. They are granted automatically at the time of installation.

Permissions in Android are coarse-grained, for example, `INTERNET` permission does not have capability to restrict access to particular Uniform Resource Locator (URL) domain(s). Also `READ_PHONE_STATE` permission allows to check if phone is ringing or in hold, at the same time it allows to read phone identifiers. Permissions like `WRITE_SETTINGS`, `CAMERA` are also similarly broad, thus violating least privilege access principle. Permissions are also not hierarchical, for example, `WRITE_CONTACTS` does not imply `READ_CONTACTS`, it must be requested separately. Same is the case with `READ_SMS` and `WRITE_SMS`. At the time of installation, user is asked to grant either all or no permissions. Often, users are unable to judge the appropriateness of certain permissions requested by apps and expose themselves to risk [33].

C. Secure System Partition

System partition of smartphone contains Android's kernel, system libraries, runtime, framework and applications [30]. Android makes system partition read-only to protect unau-
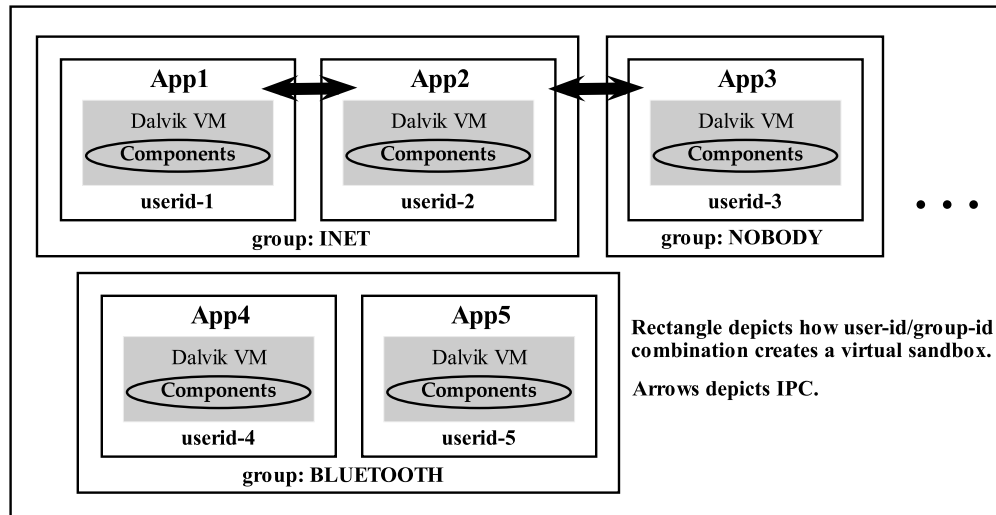

Fig. 5: Android Apps within Sandbox at Kernel-level [30]

thorized access and modification. Also, some part of file-system such as application cache and sdcard are secured with appropriate privileges to prevent tampering from adversary when device is connected with the PC using USB port.

D. Secure Google Play Store

Google discourages users to install apps from sources other than its own Play Store due to security reasons. Before making any app available for users to download, Google verifies it with Bouncer, a dynamic analysis service that executes app in a sandboxed environment to ascertain its normal behavior. Bouncer, if not invincible [34], is a reasonably effective security mechanism. Android also provides a verification service at the time of installation for apps downloaded from other sources. Google Play also has the ability to remotely un-install an app if it is found to be malicious later [35].

E. Other Security Enhancements

SELinux has been integrated into Android since Jelly Bean 4.3 to provide greater security [36]. It imposes Mandatory Access Control (MAC) policies over the traditional Discretionary Access Control (DAC) on the device. In DAC, it is the owner of the resource who decides which other interested subjects can access it, in contrast, in MAC, it is the system (and not the users) that authorizes subjects for accessing the resource. Thus, MAC has the potential to prevent any malicious activity even though root access has been compromised. This reduces the effect of kernel-level privilege escalation attacks, attempted by taking advantage of vulnerable processes that run with root privilege.

Information on the device can also be ex-filtrated by connecting it to a PC through USB using Android Debug Bridge (ADB) driver. ADB driver is created mainly for the debugging purposes, through which one could install/uninstall apps, read system partitions etc. even though device is locked prior to Jelly Bean 4.2.2. To prevent unauthorized access, Android authenticates any ADB connection using RSA keypair [37]. Also, Android prompts user for allowing access to the device through ADB connection on the device's screen, so if device is locked, attacker would not be able to gain access to that device.

Android has also removed `setuid()`/`setgid()` programs [37] which were vulnerable for some time as many root exploits had been written based on them.

Many independent Android security enhancement mechanisms have been proposed [38] [39] [40] [41]. These mechanisms allow an organization to create finer grained security policies for their employees' devices. Various context information such as phone location, installed apps' permissions and inter-app communication can be actively monitored and verified against their corresponding policies. Scope of this paper is to investigate issues related to Android malware and we do not aim to examine these prevention mechanisms.

IV. ANDROID SECURITY ISSUES

This section briefly walks through the issues that are important for user and device security in general.

A. Update Problem

Android Open Source Project (AOSP), led by Google, upgrades and maintains Android source-code. But releasing new versions/updates to the end-users remain the responsibility of Original Equipment Manufacturers (OEMs) or wireless carriers. Individual OEM branches out newer version of Android and customizes it accordingly. In some countries, wireless carriers also customize the OEM's version to suit them. This whole update chain process takes months before it finally reaches the end-users. This phenomenon is known as *Fragmentation* problem, where different versions of Android remain scattered among consumers. Specifically, handsets with older and un-patched versions may be vulnerable to publicly available exploits.

Updates for Android OS are seemingly frequent compared to their PC counterparts as there have been 25 stable releases since September 2008 [42]. Over The Air (OTA) new version update significantly changes the existing version by adding and modifying large number of files across Android platform, ensuring integrity of existing user data and apps [43]. New version update is facilitated through a service called Package Management System (PMS). Luyi Xing et al. [43] performed a comprehensive study of pileup vulnerabilities that can be exploited by malware apps in case of new version upgrades. For example, an app for older version can declare dangerous permissions in `AndroidManifest.xml` that have been introduced in next version(s). During the update process, Android does not ask user to verify newly active permissions in that existing app and grants them automatically [43]. Thus, it compromises security of the device.

B. 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. Native code has the potential to execute publicly available root-level exploits across older Android versions [19] [44] [45].

C. Types of Threats

Even though AOSP is committed to provide a secure smartphone OS, it is susceptible to social-engineering attacks, using which malicious apps can perform many undesirable activities. Following is a list of malicious activities that have repeatedly happened or can happen across Android versions.

- *Privilege escalation* attacks are done by leveraging publicly available kernel-level vulnerabilities [46] to gain root access of the device. It can also happen by exploiting one or more vulnerable components of an app that makes use of dangerous permission(s) granted to it.
- *Privacy leak or personal-information theft* happens when users grant dangerous permissions to malicious apps that read sensitive data and ex-filtrate them without users' knowledge or consent.
- Malicious apps can also *spy* on users by monitoring calls, SMS/MMS, bank mTANs, recording audio/video without users' knowledge or consent.

- Malicious apps can earn money by making calls or subscribing via-SMS to premium-rate numbers, without users' 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 malicious apps.
- *Colluding* attack happens when 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 all others. Collectively, these apps can perform malicious activities, while individually they may seem perfectly normal. For example, an app with `READ_SMS` permission can read SMSes and ask its related app with `INTERNET` permission to ex-filtrate them.
- *Denial of Service (DoS)* attack can happen when app(s) overuses already limited CPU, memory, battery and bandwidth resources, blocking users from using the device.

V. REPORTED ANDROID MALWARE THREAT PERCEPTION

Figure 6 shows the time-line of some notable malware families of Android during 2010-2013. Among them, SMS Trojans has a major contribution, some of which have infected even Google Play [48]. Large number of malicious apps have also exploited root-based attacks such as *rage-against-the-cage* [19], *gingerbreak* [45] and *z4root* [44] to gain superuser privileges for availing control of the device. Recent addition in the exploitation techniques is the *master-key* attack [49], which has left devices from Android version 1.6 to JellyBean 4.2.2 vulnerable.

Quarterly, each anti-malware company reports about Android malware threats [50] [51]. These companies also differ in the approximation of malware infection-rate among Android users. In particular, Lookout Inc. reported that the global malware infection-rate by likelihood is 2.61% for its users [52]. Moreover, two independent research also estimated real infection-rate. Lever et al. [53], used the Domain Name Resolution (DNS) traffic of smartphones in United States and found 0.0009% infection rate. Very recently, Truong et al. [54] instrumented a well-known Carat app [55] to estimate infection-rate for three malware datasets. They found infection-rate 0.26% and 0.28% for McAfee and Mobile-Sandbox dataset respectively. Therefore, at present the threat perception on Android malware is wide ranging and house remains divided in numbers.

In the following, we discuss Android malware classified accordingly to their characteristics.

A. Trojan

Trojans masquerade themselves as benign apps, but they perform harmful activities without consent or knowledge of the users. Trojans may leak confidential information of the user to outside, or they may "phish" the user to provide sensitive information such as passwords. Till second quarter of 2012, majority of variants belonged to the SMS Trojan family. Apps of this family can send messages to premium rate numbers without consent and thus incurring financial loss to the user. Apart from that, they also leak contacts, messages, IMEI/IMSI numbers to unknown domains. FakeNetflix [56] masquerades itself as popular Netflix app, phishing the user to enter their login credentials. Fakeplayer [57], Zsone [51] and Android.Foney [58] are other examples of Trojan, which incur financial loss to the user.

Due to increase in mobile banking transactions, malware authors have targeted two-factor authentication used by mobile banking firms. After capturing username and password of bank accounts using social engineering attack, Zitmo and Spitmo Trojans listen for mTANs (Mobile Transaction Authentication Numbers) to silently complete transactions [59].

B. Backdoor

Backdoor allows entry to the system bypassing all security procedures and facilitates installation of other malicious apps into the system. Backdoor generally uses root-level exploits to gain superuser privilege, so that it can hide itself from other security apps, or worse it may disable them also. Number of root-level exploits have been leveraged such as *rage-against-the-cage* [19] and *gingerbreak* [45] to gain full-control over the device. Basebridge [48], KMin [48], Obad [18] are example of well-known backdoor apps.

C. Worm

Worm app can create an exact or similar copies of itself and spreads them through network or removable media. For example, Bluetooth worms can send copies to other devices via Bluetooth automatically. *Android.Obad.OS* [18] is one such example that can spread malicious apps via Bluetooth.

D. Botnet

These type of 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 as causing a denial of service attack. *Bot* can also include commands to download malicious payloads automatically. Geinimi [48], Anserverbot [48], Beanbot [48] are well known examples of 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. to perform wrongful actions at later stage. They may also send all the collected information to the remote server. Nickyspy [48], GPSSpy [51] are known examples of spyware.
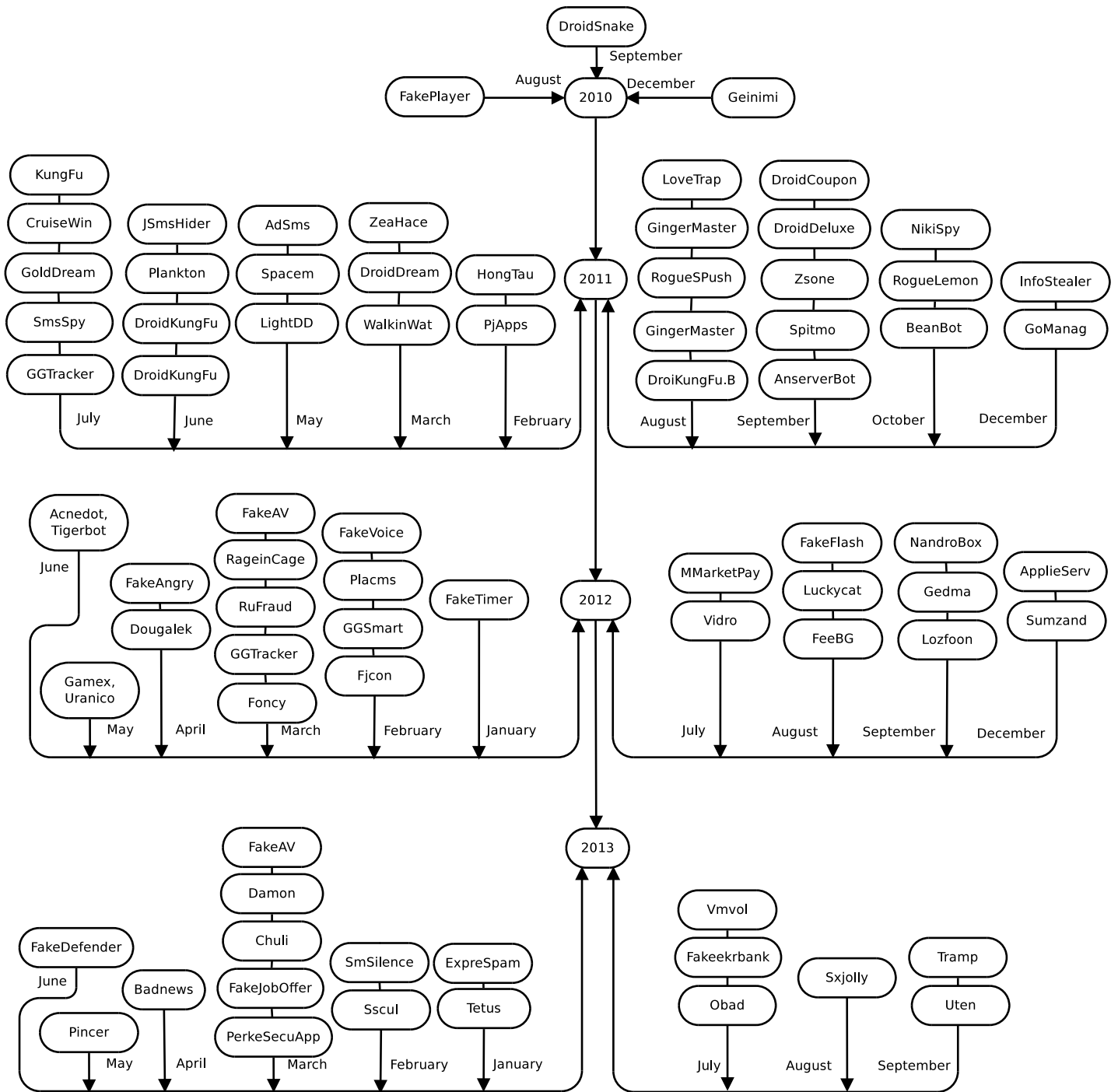
Fig. 6: Android Malware Family Chronology [47] [15] [14] [10] [12] [11]

F. Aggressive Adware

Android provides coarse and fine grained location services. Shady ad affiliate networks misuse them to bombard users with personalized advertisements. Notable behavior of this type of apps is the creation of shortcuts on home-screen, bookmarks, changing default search engine settings, push unnecessary notifications etc. which hinders user's effective use of the device. Plankton [51] is an example of well known aggressive adware.

G. Ransomware

This type of app locks the device and makes it completely inaccessible until some ransom amount is paid through online payment service. For example, *FakeDefender.B* [60] malware, after installation, shows fake malware alerts masquerading as *avast!* [61] antivirus. After that, it locks the device and asks user to pay a ransom amount to remove threats and unlock it.

VI. MALWARE PENETRATION TECHNIQUES

In this section, we summarize penetration techniques used by Android malware.

A. Repackaging Popular Apps

Repackaging is a process of ripping-off popular free/paid apps from one app-store, adding malicious payload into them, and distributing them through other third-party app-stores. App is repackaged using reverse-engineering tools and techniques. Repackaging process is illustrated in the Figure 7. In the following, we report the main steps involved in app repackaging:

- Find popular free/paid app from some renowned app-store and download it on PC.
- Disassemble the app using tools such as *apktool* [62].
- Write malicious payload either directly in Dalvik bytecode, or in Java and then convert it into Dalvik bytecode using *dx* [63] tool.
- Add that payload into benign app. Make necessary changes in `AndroidManifest.xml` and *resources*, if any.
- Assemble modified source again using *apktool*.
- Finally, distribute repackaged app by self-signing with another certificate to other app-stores for free.

Repackaging is one of the most common technique to spread malicious activities. More than 80% samples among Malware Genome Dataset are repackaged ones [9] from the legitimate apps. This technique can be repeated for any number of popular apps, or can add different malicious payload each time using the same app. This technique can also be used to generate number of variants of existing malware. As signature of each malware variant gets changed, anti-malware solution with fixed number of signatures cannot identify them. Repackaging is one of the biggest threats as it can pollute the centralize distribution system and financially hurts the original developer. Rogue elements can even repackage apps for diverting advertisement revenues.

AndroRAT APK Binder [64] tool can repackage and Trojanize any legitimate app automatically by adding Remote Access functionality as a payload. Adversary from remote location, using GUI, can make infected device send SMS messages, make phone-calls, access device location, record video or audio and access files from device's storage.

B. Drive-by Download

An attacker can employ social engineering, aggressive advertisements and click-jacking attacks to make user mistakenly download malicious apps. As soon as user visits a malicious URL, it downloads a malicious app automatically, optionally may disguise itself a legitimate one to be able to get permission from the user to install itself. *Android/NotCompatible* [21] is a notable example of drive-by download attack.

C. Dynamic Payload

An app can also embed malicious payload in the form of `apk`/`jar` file, either encrypted or in plain format, into the resources. After the installation, when app executes, it optionally decrypts the payload. If payload is in the form of `jar` file, then using the `DexClassLoader` API, it loads payload into Dalvik VM dynamically to execute it. Otherwise, it will ask the user for confirmation to install the embedded `apk` by disguising itself as some important update. App can also execute native binaries using `Runtime.exec` API, which is roughly equivalent to `fork()`/`exec()` in Linux. BaseBridge [48] and Anserverbot [48] malware families adopt this technique. Some malware families does not embed malicious payload as a resource, but rather download it from a remote server. DroidKungFuUpdate [48] is one such example of dynamically executing payload from a remote server. This type of penetration technique is very difficult to detect using fixed signature based or static analysis methods.

VII. STEALTH MALWARE TECHNIQUES

Android works under the condition of low processing power and limited memory as well as battery availability. Anti-malware apps cannot perform real-time deep analysis of apps due to above constraints, unlike PC counterpart. Malware authors view these inabilities of anti-malware apps as an opportunity to make their malicious payload highly obfuscated to hide themselves from anti-malware signatures. 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.

Code obfuscation is evolving on Android platform as well [65] [66], following the trends of their PC counterparts. Obfuscation techniques in general are employed for one or more of the following purposes.

1) To protect proprietary algorithm within app from rivals by making reverse-engineering very difficult.
2) To protect Digital Rights Management of multimedia resources in order to reduce piracy.
3) Developers perform obfuscation on their apps to make them more compact and thus faster.
4) Malware authors use obfuscation to hide itself from anti-malware scanners and deep-analysis for a longer duration, so that it can propagate and infect more and more devices.
5) Prevent or at least delay human analysts or automatic analysis engines from figuring out the intention of malicious code.

As Android app contains Dalvik bytecode, it is amenable to reverse-engineering due to type information available such as class/method types or definitions, variables, registers and literal strings along with instructions. Code transformation techniques are applied on Dalvik bytecode to optimize it, for example, Proguard [67] is an Android as well as Java obfuscator. Proguard is an optimization tool to remove unused classes, methods and fields. Meaningful class, method, field and local variable names are replaced with smaller names to make it difficult to understand their purpose. Dexguard [68] is a commercial Android code protection tool. Advanced code obfuscation techniques like class encryption, method merging, string encryption, control flow mangling are employed with Dexguard to protect app from reverse-engineering.
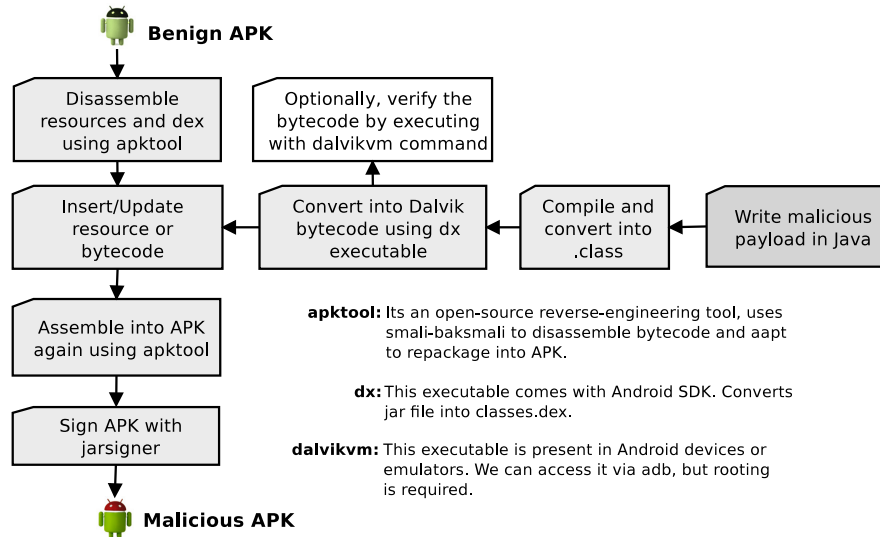
Fig. 7: App Repackaging Process

This section covers various code transformation methods employed by malicious apps to make itself obfuscated and generate large number of variants. In fact, code transformation can also be applied in such a way to make disassembling erroneous [69].

A. Junk Code Insertion and Opcode Reordering

Junk code or *no-operation* code (nop) is a well-known technique used to change executable size and evade anti-malware signatures. Inserting junk code in the app preserves the semantics, but changes the opcode sequence. Opcode can also be re-ordered by putting *goto* instructions in-between to preserve the original execution. These methods can be successfully used to fool the signature-based or opcode-based detection solutions [65] [66].

B. Package, Class or Method Renaming

Every Android app on app-store is generally identified by a unique package name. Dalvik bytecode also contains names of all classes and methods in it. Some malware signature may depend upon the package, class or method names of some malicious app for detection [70]. Even though trivial, by changing those names, a malicious variant can evade detection [66].

C. Altering Control-flow

Some anti-malware may generate signatures by analyzing control-flow graphs (CFGs) of malicious apps in order to detect them [70]. CFGs can be modified by simply adding unnecessary *goto* instructions or by inserting and calling junk methods. Even though trivial, this technique can successfully evade anti-malware signatures [66].

D. String Encryption

Literal strings within a program such as messages, URLs and shell-commands reveal a great deal about an app under inspection. To prevent analysis, those strings can be encrypted in order to render them unreadable. Also, each time string encryption is applied, different encryption methods (or keys) can be used to make it difficult to automate decryption. In that case, literal strings can only available during execution, thus frustrating static analysis methods.

E. Class Encryption

Important code such as license-checks, paid downloads and DRMs can be hidden by encrypting entire classes those deal with them [68].

F. Resource Encryption

Content of *resources*, *assets* and native libraries can be encrypted, modifying their access code in order to decrypt them at runtime [68].

G. Using Reflection APIs

Static analysis methods mainly look for Android API calls within malicious apps to inspect their behavior. Java reflection allows us to programmatically create class instances and/or method invocation using literal strings. To identify exact class or method names, data-flow analysis must be leveraged. But, because those literal strings can also be encrypted, as we have seen before, it rather becomes impossible to automatically find those API calls, hindering the static analysis.

VIII. APPROACHES FOR ASSESSMENT, ANALYSIS AND DETECTION

Android security solutions such as vulnerability assessment, analysis and malware detection are broadly divided into two types: 1) Static; 2) Dynamic. Static methods are quick, but it has to deal with false-positives carefully. Dynamic methods, though time-consuming, are very helpful when apps are highly obfuscated. Hybrid approaches those leverage both static as

well dynamic methods are also exist due to the limitations of both.

Security solutions can be either rule-based [71] or they can extract features to create a machine-learning model [72]. Inappropriate feature selection can degrade the performance of model, generating false-positives (i.e., false detection of benign apps). Moreover, the number of features under problem should be small and effective, in order to make solution feasible to use in real-time. Feature reduction methods along with strong statistical measures such as mean, standard deviation, chi-square, haar transforms can help us identify prominent features. Learning model can then be created by giving above features clustering or classification algorithms.

A. Static Approach

Static analysis based approaches work by just reading or disassembling apps under consideration, without executing them. They can miserably fail against various obfuscation techniques we covered in Section VII.

1) Signature-based Malware Detection: Most of the current Android anti-malware products use this efficient approach for malware detection on device. It can extract interesting syntactic or semantic features [73] to find signature(s) that matches with existing database. Signature-based methods becomes ineffective if variants of existing malware are generated through polymorphism. Moreover, fast signature generation and its distribution becomes very important in the times of malware outbreak. Parvez et al. [74] devised a prototype, called AndroSimilar, to automatically extract signatures and detect zero-day variants of existing malware.

2) Component-based Analysis: In order to perform detailed app-security assessment or analysis, an app can be disassembled to extract important content such as `AndroidManifest.xml`, resources and bytecode. Manifest file contains important meta-data about an app in question, such as list of components (i.e., Activities, Services etc.) and required permissions. App-security assessment solutions can analyze components using their definition and interaction within bytecode to find out vulnerabilities [3] [75] [76].

3) Permission-based Analysis: Permission to access sensitive resource is the central design point of Android security model, so that no application, by default, has any permission that affects user's security. By just looking at permission requests, it is not possible to decide whether an app is malicious, but nevertheless, it is an important feature to assess the risk associated by granting them all [77] [78].

Sanz Borja et al. [79] used `<uses-permission>` and `<uses-features>` tags present in `AndroidManifest.xml` file as features. They applied machine learning algorithms, such as Naive Bayes, Random Forest, J48 and Bayes-Net on a dataset consisting of 249 malicious and 357 benign apps. Huang Chun-Ying et al. [80] has also used requested permissions apart from other manifest features, then applied machine learning algorithms on a dataset consisting of 1,25,249 malicious and benign apps. Enck et al. [81] developed certification tool, called Kirin, used to define set of rules regarding the combination of certain permissions requested by an app.

4) Dalvik Bytecode Analysis: Dalvik bytecode contains useful, semantically rich, type information such as classes, methods and instructions. We can utilize them to verify app's behavior. Detailed analysis using control-flow, data-flow can throw light upon some of the dangerous functionality performed by apps such as privacy leakage and telephony services misuse [26] [71] [82]. Control-flow and data-flow analysis may also help de-obfuscated bytecode, for example, usage of Java Reflection API [83].

Control-flow analysis of bytecode helps in identifying possible paths that can follow during execution. Dalvik bytecode contains jump, branch and method invocation instructions that alter the order of execution. To facilitate further analysis, we can generate an intra-procedural (i.e., spans single method) or inter-procedural (i.e., spans across methods) control-flow graph (CFG) of the bytecode. Karlsen et al. [83] formalized the Dalvik bytecode in order to perform control-flow analysis on it.

Data-flow analysis of bytecode helps in predicting possible set of values at some point of execution. We can use CFG to traverse possible execution paths. In the same way as control-flow, we can perform data-flow analysis either at intra-procedural or inter-procedural level, among which latter one improves approximation of the desired output. In particular, we can perform a special type of data-flow analysis, called constant propagation, to find constant arguments of some sensitive API calls during execution. Consider for example, a malicious app that sends premium SMSes, if that app sends SMSes to some hardcoded numbers, using constant propagation data-flow analysis they can be retrieved [71]. Another special type of data-flow analysis, called taint analysis, tracks the variables that hold some important information. For example, taint analysis can identify privacy leakage within apps [82].

We can also utilize API-calls within bytecode to identify malicious behavior [84] as well as similar apps [85]. Zhou et al. [9] utilized just sequence of opcodes within Dalvik bytecode instructions to catch repackaged (i.e., similar) apps.

5) Re-targeting Dalvik Bytecode to Java Bytecode: Availability of number of Java decompilers [86] [87] [88], as well as static analysis tools based on it [89] [90] [91], has motivated some researchers to re-target Dalvik bytecode to Java bytecode. Enck et al. [92] developed *ded* tool that can convert Dalvik bytecode to Java. Later, they performed static analysis on Java such as control-flow, data-flow, using Fortify SCA [91] framework. Octeau et al. [93] developed *Dare* tool to convert Dalvik bytecode to Java bytecode with nearly 99% accuracy. Bartel et al. [94] developed *Dexpler* plugin for static analysis framework called Soot [89]. *Dexpler* converts Dalvik bytecode into Soot's internal Jimple code, however, it is unable to handle optimized *dex* (odex) files. Gibler et al. [95] employed *ded* and *dex2jar* [96] to convert Dalvik bytecode into Java and Java bytecode respectively, and used static analysis framework called WALA [90] to identify privacy leakage within Android apps at large scale.

B. Dynamic Approach

Although static approaches for analyzing apps are quick, they fall short of detecting encrypted and new malware to

much extent. And here comes the role of dynamic approaches, where we execute the app in a protected environment, providing all the emulated resources it needs, making it feel at home, thereby learning its malicious activities. Although dynamic analysis on Android seems to be very new, there exists notable research in this field. As, execution of apps in Android is event based, it is important to trigger those events. UI gestures such as tap, pinch, swipe, keyboard and back/menu key press must be automatically triggered, in order to make app perform various activities. Android SDK comes with a tool, called *monkey* [97], to automate some of the above gestures. In order to perform in-depth monitoring of an app, one may need to change some part of the Android OS, this technique is 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 particular time of the day. Also, other anti-emulation techniques such as detecting sandbox environment [34] and performing malicious activities after some time delay can successfully evade dynamic analysis. Dynamic approaches can be broadly divided into following three categories.

1) Profile-based Anomaly Detection: Malicious apps may perform Denial of Service (DoS) attack by over utilization of constrained hardware resources. Range of parameters at different layers of Android subsystem are collected, such as CPU load, memory statistics, network traffic, battery usage and system-call access sequence, from benign as well as malicious apps. After that, machine learning methods are applied on the collected data in order to distinguish abnormal behavior [72] [98].

2) Malicious Behavior Detection: Specific malicious behaviors like sensitive data leakage, sending SMS/emails, making calls without user's consent can be accurately detected by tapping or monitoring those particular areas of interest [99] [100].

3) Virtual Machine Introspection: The downside of monitoring app behavior within emulator (VM) is that emulator itself is susceptible of being compromised by a malicious app, there by defeating our purpose. To counter this, Virtual Machine Introspection approach can be employed, in which, behavior of apps is observed external to the emulator [101].

IX. 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 to completely off-device.

A. On-Device

Signature-based malware detection is provided as apps by many anti-malware companies due to its efficiency and simplicity. But, as noted before, detailed assessment as well as analysis is difficult to perform on device, because of constraints within smartphones and Android subsystem. Lightweight risk assessment by analyzing components and permissions can also be done on smartphones [81]. Following

are some of the limitations of anti-malware apps for providing robust protection within devices.

- Anti-malware apps run as any other normal app without special privileges. As a result, they are also under the purview of process isolation. Because of this, they cannot directly scan other app's memory and private files for malware detection.
- Though Android allows running background services of apps, it can stop anti-malware app services while running out of resources. Even other apps can stop an anti-malware app from executing, if they have appropriate privileges.
- Without acquiring *root* privileges, anti-malware app cannot create system hooks for monitoring file-system or network access.
- Without acquiring *root* privileges, anti-malware app cannot uninstall other malicious apps automatically, it must rely on users to uninstall them.

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

Though quick assessment or detection can be performed on the device itself, detailed and computationally expensive analysis can be performed at remote server. Continuous availability of bandwidth and its cost is a concern, but at the same time signature database in the smartphone can be greatly reduced to make anti-malware app limited-resource friendly. In the case of profile-based anomaly detection, resource usage as well as other parameters can be collected at client-side to send them to remote server for detailed analysis, and finally, results can be sent back to the device.

C. Off-Device

It is important to automate deep analysis of new malware samples, so that human analysts can quickly understand them and find mitigation solutions. This type of automated deep analysis solutions require more computational power as well as memory. Because of this, they are usually deployed off-device [71] [26] [101] [82] [72].

X. State-of-the-Art Tools for Android App Assessment, Analysis, and Detection

Industry and academia have proposed several solutions for analysis and detection of Android malware. 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 malware detection; 2) Methodology as discussed in Section VIII; 3) Deployment as discussed in Section IX.

A. Reverse-Engineering Tools

Content of Android package (APK) is stored in a binary format for efficiency. Before assessment, analysis or detection task begins, it is important to disassemble it to make further processing easier. There are number of tools available to

disassemble or de-compile APK files. In the following, we note some of the popular reverse-engineering tools.

1) *apktool* [62] can decode binary content inside APK into nearly original form in project-like directory structure. It disassembles binary resources and converts bytecode within `classes.dex` into smali [102] syntax for easier reading as well as manipulation. After making any changes, it can also repackage it back into APK. This tool seems to be the best among all reverse-engineering tools.

2) *dex2jar* [96] is a disassembler that can parse both `dex` and optimized `dex` file, providing a light-weight API to access it. *dex2jar* can also convert `dex` file into a `jar` file, by re-targeting Dalvik bytecode into Java bytecode, for further manipulation. Moreover, it can also assemble back `jar` into `dex` after modification.

3) *Dare* [103] 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 vast range of traditional techniques developed for Java applications, including de-compilers. Octeau et al. [93] demonstrated that *Dare* is nearly 40% more accurate than *dex2jar*.

4) *Dedexer* [104] disassembles `classes.dex` into Jasmin-like syntax, by creating separate file for each class, along with package directory structure, for easier reading and manipulation. But unlike *apktool*, it cannot assemble back those intermediate class files.

5) *JEB* [105] is a leading professional reverse-engineering software for Android apps, available for Windows, Linux and Mac platforms. It is a GUI-based interactive de-compiler for security analysts to see content of Android apps, such as manifest, resources, certificates, literal strings and examine its decompiled Java source by providing easy navigation through cross-references. JEB converts Dalvik bytecode directly into Java source by better utilizing semantic information present in Dalvik bytecode, without going through Java bytecode. Exceptionally, JEB can also de-obfuscate Dalvik bytecode to make disassembled code more readable in comparison to its counterparts [96] [62]. JEB also supports Python scripts or plugins by allowing access to decompiled Java code's Abstract Syntax Tree (AST) through API. This feature is helpful in automating particular analysis needs. According to us, it is the best reverse-engineering tool so far.

B. Androguard

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

Androguard [70], an open-source, static analysis tool, can disassemble and decompile Android apps to make reverse engineering much easier. It can generate control flow graphs for each method and provides access of all through Python-API, as well as graphic formats. Androguards unique Normalized

Compression Distance (NCD) approach can find similarities and differences in code between two apps reliably, which can also be used to detect repackaging.

It provides a rich API in Python to access disassembled resources and static analysis structures like basic-blocks, control-flow and instructions of an APK, using which, one can develop their own static analysis methods. Following are some of features explained briefly.

1) Code Similarity Among Apps: Androguard finds similarities between two apps by calculating Normalized Compression Distance between each method pairs and calculates a score from 0-100, where 100 means apps are identical. It displays IDENTICAL, SIMILAR, NEW, DELETED and SKIPPED methods of first app with respect to another one. In the same way, it can also display 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 Common Subsequence algorithm on these strings of two basic blocks to find differences between them [106].
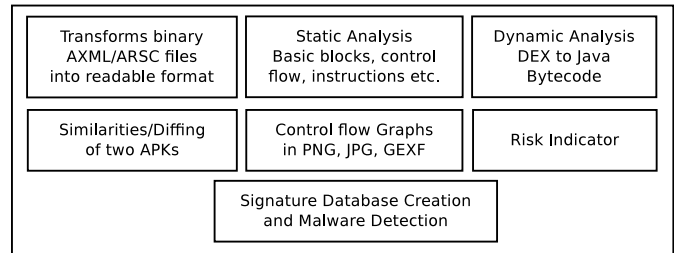

Transforms binary AXML/ARSC files into readable format	Static Analysis Basic blocks, control flow, instructions etc.	Dynamic Analysis DEX to Java Bytecode
Similarities/Diffing of two APKs	Control flow Graphs in PNG, JPG, GEXF	Risk Indicator
Signature Database Creation and Malware Detection		

Fig. 8: Features of Androguard

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 Generation and Detection for Malicious Apps: Androguard manages a database of signatures and provides an interface to add/remove signatures to/from the database. Signature is described in a 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 [72], 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 can be seen in the Figure 9, 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 Feature Manager. Feature Manager may also perform some pre-processing on the raw feature data.
- *Processor:* It is an analysis and detection unit. Its role is to receive the feature vectors from Main Service, analyze them and perform threat assessment 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 device infection. Alert Manager smoothes the result to avoid 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 processors and receiving final recommendation from alert manager. Loggers can log information for de-

bugging, calibration and experimentation. Configuration Manager manages the configuration of the 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 which can result in activation/deactivation of processors and feature extractors. This change in operation modes is resulted due to change in resource levels.

- *Graphical User Interface:* It interacts with 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)

Parvez et al. [74] 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) [107] used in digital forensics to identify similar documents.

Intuitively, completely unrelated apps should have lower probability of having common features. When two unrelated
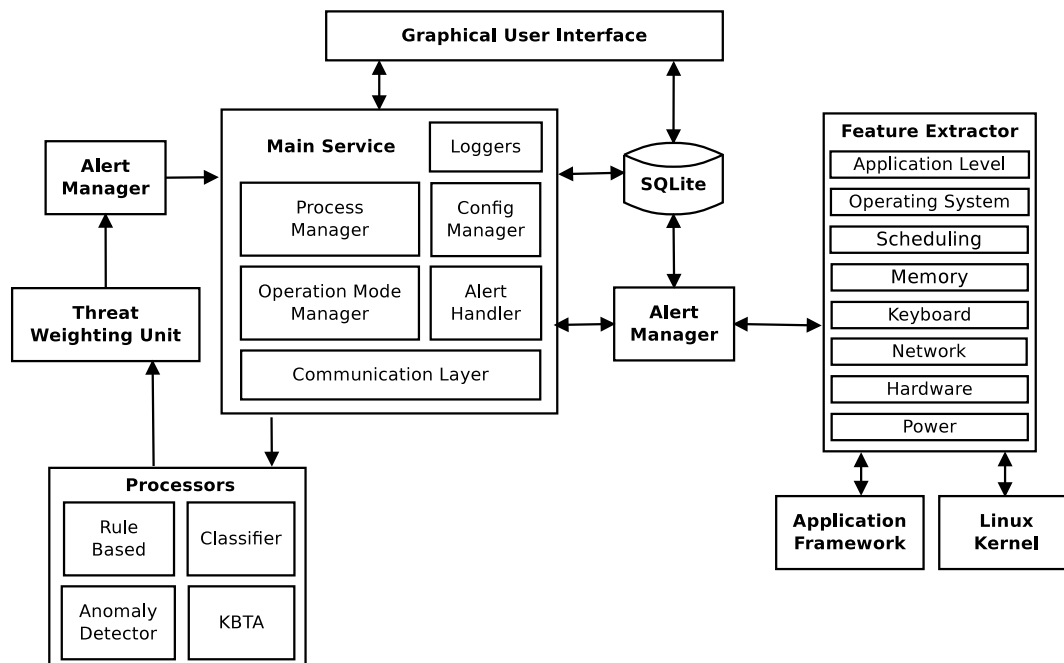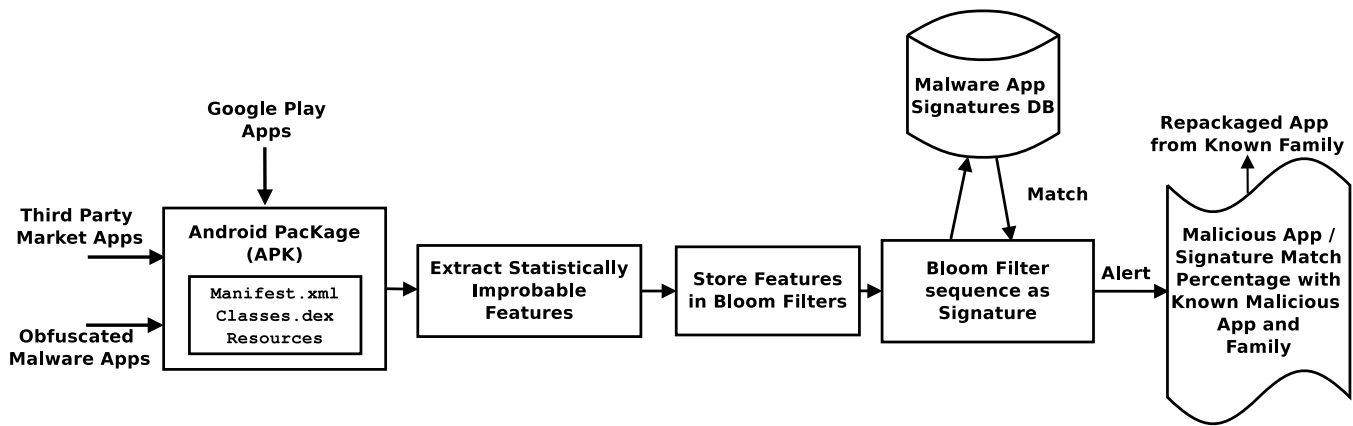

Fig. 9: Architecture of Andromaly

Fig. 10: AndroSimilar Methodology

apps share some features, such features should be considered weak as using these shall lead to false positives [108]. 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 [107]. Figure 10 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 to 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 the representative database. If similarity score of an unknown app with any existing family signatures matches beyond a threshold, then it is labeled as malicious.

E. Andrubis

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

Andrubis [109] is a web-based malware analysis platform, build upon some well-known existing tools. Users can submit suspicious apps through web interface. After analyzing app on the remote-server, Andrubis then returns detailed static as well dynamic analysis report of the same. Andrubis also provides a rating for app's behavior between 0-10, where 0 means benign and 10 means malicious. It is built upon Droidbox [110], TaintDroid [99], apktool [62] and androguard [70].

F. APKInspector

Goal: Malware Analysis
Methodology: Static

Deployment: Off-Device

APKInspector [111] is a full-fledged static analysis program for Android apps, combining some of the well-known tools like *Ded* [112], *smali/baksmali* [102], *apktool* [62] and *Androguard* [70]. It provides a rich GUI and provides following features:

- Meta-data about app
- 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 of *smali* code

G. Aurasium

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

Aurasium [113] is a powerful technique that takes control of execution of apps, by enforcing arbitrary security policies at runtime. To be able to do that, Aurasium repackages the Android apps to include code for policy enforcement. Aurasium's Security Manager component can apply policies not only at individual app level, but across multiple apps too. Any security and privacy violations are reported to the user. Thus, it eliminates the need of 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.

Limitation of Aurasium is that currently it is not stealthy, that means it can be detected by apps due to change in app signature, as well as presence of its native library. Thus, app may not reveal its malicious behavior, and hence avoiding detection. As Aurasium depends on repackaging, it may altogether fail to disassemble (or assemble) an obfuscated 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, for testing apps uploaded by third-party developers, before allowing users to download them. 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. [34] presented their analysis of Bouncer environment by implementing a custom command and control app. Dynamic code loading techniques have helped evade scrutiny from Bouncer [114].

I. CopperDroid

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

Reina et al. proposed CopperDroid [98], 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's 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 [115].

J. Crowdroid

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

Crowdroid [100] is a behavior based malware detection system. 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.

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 that crowdsourcing app should be kept running in the background for monitoring, which can drain the resources. Also, this technique is yet to be tested on wide varieties of malware families available, to check its robustness.

K. Droidbox

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

Droidbox [110] is a dynamic analysis tool based on Taint-Droid [99] using modified Android framework for API call analysis. Figure 12 shows static and dynamic analysis operations performed within Droidbox. App analysis begins with the static-pre-checking, which includes parsing permissions, activities and receivers. The app under analysis is executed

App-log
- Device info
- App list
- System calls

Crowdroid
Remote Server

Central Database

Devices running
Crowdsourcing app
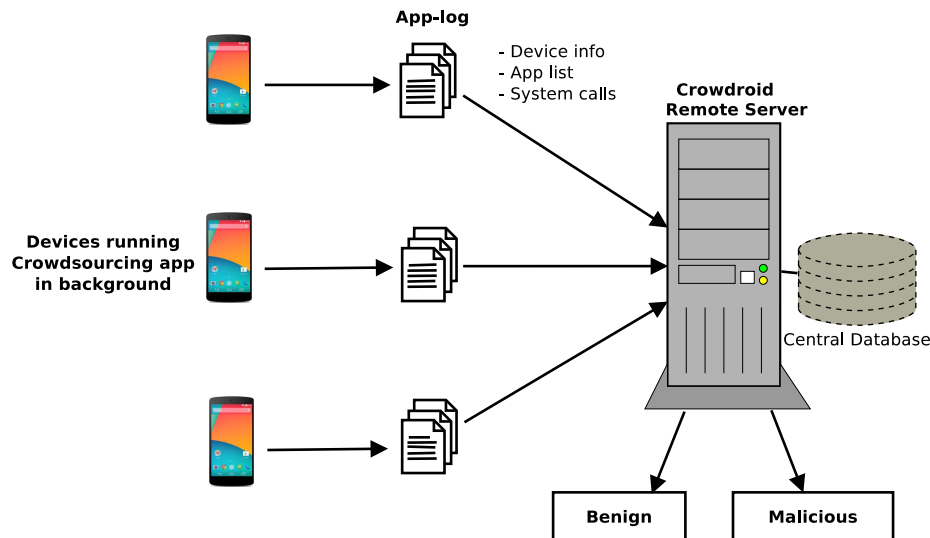in background

Benign   Malicious

Fig. 11: Crowdroid Architecture

in emulated environment to perform taint-analysis and API monitoring. Taint-analysis involves labeling (tainting) private and sensitive data that propagates through 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 app that is responsible for transmission. API monitoring involves API logging with its parameters and return value. The results generated consist of following:

- Hashes of analyzed apps
- 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 with Android API
- Temporal order of operations
- Tree-map for similarity analysis

STATIC OPERATIONS

Prechecking
- Permissions
- Activities
- Receivers

DYNAMIC OPERATIONS

Taint Analysis
Finds data leaks via
- Network
- File
- SMS

API Monitoring
Logging and Analyzing
- API
- Parameters
- Return value
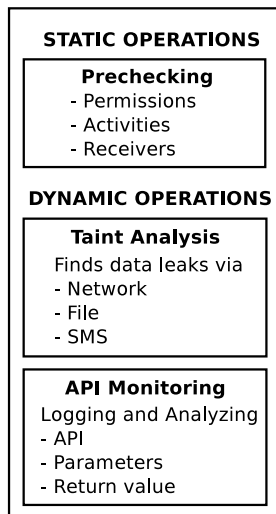
Fig. 12: Features of Droidbox

Limitation of Droidbox is that it monitors only tasks performed within the Android Framework. If native code leaks data, it will get unnoticed.

L. DroidMOSS

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

DroidMOSS [9] is a prototype that detects app repackaging using semantic file features. More specifically, it extracts DEX opcode sequence from an app, then generates a signature from it using fuzzy hashing [116] technique. It also adds developer certificate information, mapped into unique 32-bit identifier, into signature. Signatures of two apps are compared with edit-distance algorithm for calculating similarity score. Proposed approach is discussed shown in Figure 13.

Intuition behind DroidMOSS of using just opcodes as a feature is that it might be easy for adversaries to modify operands, but much harder to change actual opcodes [9]. This approach has several disadvantages. First, as it considers only DEX bytecode, ignoring native code and resources of the app, as resources most of the time are same. Second, opcode sequence does not contain higher level semantic knowledge. Smart adversary can easily evade this technique by using obfuscations such as adding junk bytecode, method restructuring, control flow alteration, which do not contribute to the outcome of the app.

M. DroidScope

Goal: Analysis
Methodology: Dynamic
Deployment: Off-Device

DroidScope [101] is a Virtual Machine Introspection (VMI) based dynamic analysis framework for Android apps. Unlike other dynamic analysis platforms, it does not reside inside the emulator, but constructing OS-level and Dalvik-level semantics by residing outside the emulator. Hence, even the privilege escalation attacks in the kernel can be detected. It also makes the attackers task of disrupting analysis difficult. DroidScope is built upon QEMU emulator, and also provides a set of APIs to customize analysis needs to human analysts. Android malware families DroidKungFu and DroidDream were analyzed and detected successfully, however DroidScope's effectiveness against other malware families needs to be tested.

N. Drozer

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

Drozer [117] is a comprehensive attack and security assessment framework for Android devices, available both as opensource and professional version. It allows security enforcement agencies to remotely exploit Android devices in order to find vulnerabilities and threats associated with them. Figure 14 shows the working of Drozer. Following is the list of features supported by the Drozer:

- It installs an Agent app on devices that 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 such as services, content providers and broadcast receivers to find vulnerabilities in them.
- It can create a shell on devices, through which one can remotely interact with Android OS.
- It allows one to create an exploit by using known rooting vulnerability and further combine it with additional shellcode to get maximum leverage of the device.
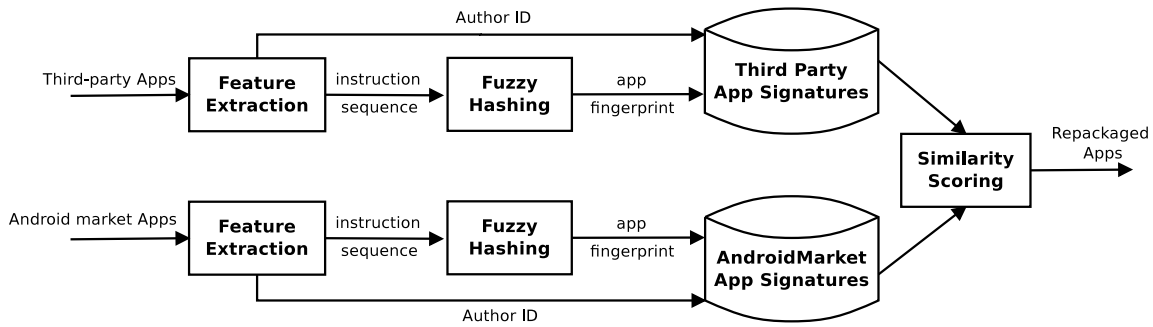
Fig. 13: DroidMOSS Methodology

O. Kirin

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

In [81] authors propose a security policy enforcement mechanism, called *Kirin*, during app installation. Kirin defines a set of rules regarding the combination certain permissions requested by an app that could prove to be harmful for the user-device. If an app fails to satisfy those security rules, installation is denied. Thus, it rigidly make decisions, based on set of rules, on-behalf of the users.

P. TaintDroid

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

TaintDroid [99] extends the Android platform to track privacy sensitive information-flow within third-party apps for leak. The sensitive data is automatically tainted (or labeled) in order to keep track whether it leaves the device. When the sensitive data leaves the system, TaintDroid records the label of data and the app which sent the data along with its destination.

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.

Lets consider working of TaintDroid with a scenario, where data of one trusted app is accessed by some untrusted app and sent over network. This is shown in the Figure 15. Firstly, the information of the trusted app is labeled according to its context. A native method is called which interfaces Dalvik VM interpreter to store taint markings in a virtual taint map. Every interpreter simultaneously propagates 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 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 app's activity as malicious.

XI. CONCLUSIONS 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
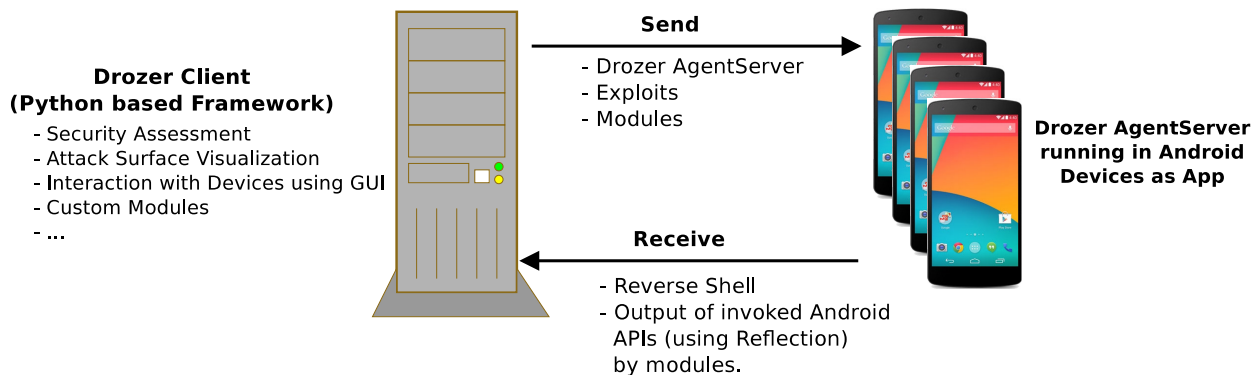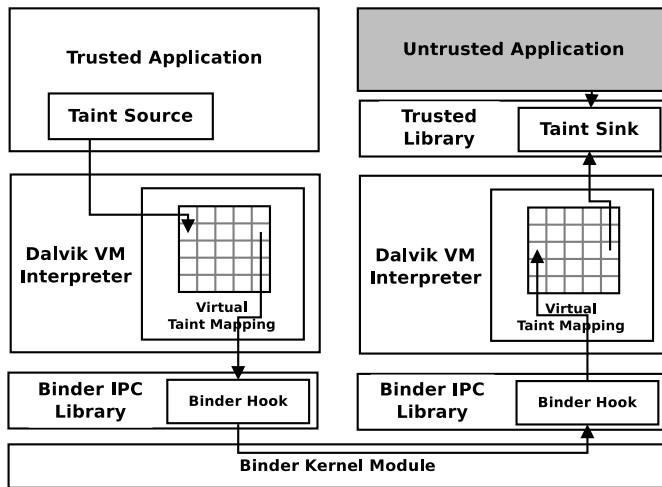

Fig. 14: Working of Drozer

Fig. 15: Taint propagation in TaintDroid

vectors to target Android platform. Due to large increase in unique malware app signature(s) and known limitations of Android security, 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 app stores to pollute online distribution mechanisms. In this review, we discuss android security and its issues, Android OS limitations, malware penetration and its implications on android ecosystem, prominent security solutions visualizing a generic analysis and detection approach.

Due to enormous popularity of Android platform among consumers and developers malware authors see an opportunity to gain monetary benefits out of them and thus android malware apps is a big threat. Between 2010 to 2013, there is an increase of hundreds of malware apps using stealth techniques to bypass commercial anti-malware products. Table I lists summary of some prominent analysis, malwaer app detection methods according to their goal, methodology and deployment. Summary shows there is no single solution that addresses each issue.

Androguard is a robust, android open source to develop custom static analysis tool(s). Andrubis and APKInspector leverages Androguard for app analysis. To study the behavior rating of Andrubis, we implemented a custom SMS based botnet, uploaded at Andrubis web service and obtained 9.9/10.0 malicious rating where as same app on VirusTotal with 47 commercial anti malware failed to detect the unseen sample. We report Andrubis' behavior rating feature is robust agaisnt zero day malware apps.

DroidMOSS and AndroSimilar employ fuzzy hashing approach for repackaged apps, new malware variant(s) detection rather than conventional signature based method. AsrRepackaging and code tranformation are easy on android platform, it is worth evaluating research directions to propose mitigation of such issues. Androguard leverages Dalvik bytecode, whereas DroidMOSS employs Dalvik opcode sequence whereas, AndroSimilar works on raw byte features. Evaluation of DroidMOSS is not possible due to unavailability of its

source code thus prototype needs to be compared against existing approaches. AndroSimilar and Androguard are tested to report that latter gives an accurate similarity score between two related apps in comparison to former method. Androguard being a semantic approach, takes long time to generate similarity score in comparison with AndroSimilar, a byte based approach. Thus, AndroSimilar is more suitable to be ported as Android app to detect unseen malware variants. We believe AndroSimilar is a promising approach against malware app variant. To tackle wide variety of new malware, a comprehensive evaluation framework incorporating robust static and dynamic methods can be proposed on Android platform.

Android malware threats are persistent due to large number of devices still running on older and vulnerable OS versions. Section VIII discusses, *static* and *dynamic* analysis approach. Both approaches can be used separately, each one has its own limitation(s). Static analysis can be thwarted by employing encryption and/or transformation techniques discussed in Section VII. Dynamic analysis can be evaded by several anti-emulation techniques covered in Section VIII-B. Manual analysis has become infeasible due to a big increase in the number of unknown malware samples.

We propose an automated, hybrid approach for Android malware analysis. Architecture of the proposed approach shown in Figure 16 is our proposed future research. As given in the diagram, APK file is dissected with static analysis. In case of its failure against encrypted code, dynamic analysis performs behavioral detection. Static and Dynamic analysis generate app activity reports to assist a malware analysts to decide upon unknown suspicious sample. Finally, we conclude that hybrid detection approach(s) are gaining prominence for analyzing malware.

REFERENCES

[1] G. Inc., Android Smartphone Sales Report, 2013, http://www.gartner. com/newsroom/id/2665715 (Online;Last Accessed March 17 2014).
[2] Number of available Android applications, http://www.appbrain.com/stats/number-of-android-apps (Online; Last Accessed 11th February 2014).

Tool	Goal			Methodology						Deployment			Availability
	Assessment	Analysis	Detection	Static	Dynamic	System call/API	Profile-based	Behavioral	VMI	On-Device	Distributed	Off-Device	
Androguard [70]	✔	✔	✔	✔								✔	Free
Andromaly [72]			✔		✔		✔				✔		Free
AndroSimilar [74]			✔	✔								✔	–
Andrubis [109]		✔	✔	✔	✔		✔	✔				✔	Free
APKInspector [111]		✔		✔								✔	Free
Aurasium [113]			✔		✔			✔				✔	Free*
CopperDroid [115]		✔	✔		✔	✔			✔			✔	Free*
Crowdroid [100]			✔		✔	✔		✔			✔		–
DroidBox [110]		✔			✔	✔		✔				✔	Free
DroidScope [101]		✔			✔	✔		✔	✔			✔	Free
Drozer [117]	✔			✔	✔						✔		Free/Paid
JEB [105]		✔	✔	✔								✔	Paid
Kirin [81]	✔			✔						✔			Free
TaintDroid [99]			✔		✔	✔		✔				✔	Free

TABLE I: Summary of Assessment, Analysis and Detection Tools for Android Platform according to their Goal, Methodology and Deployment. * indicates web-based interface.
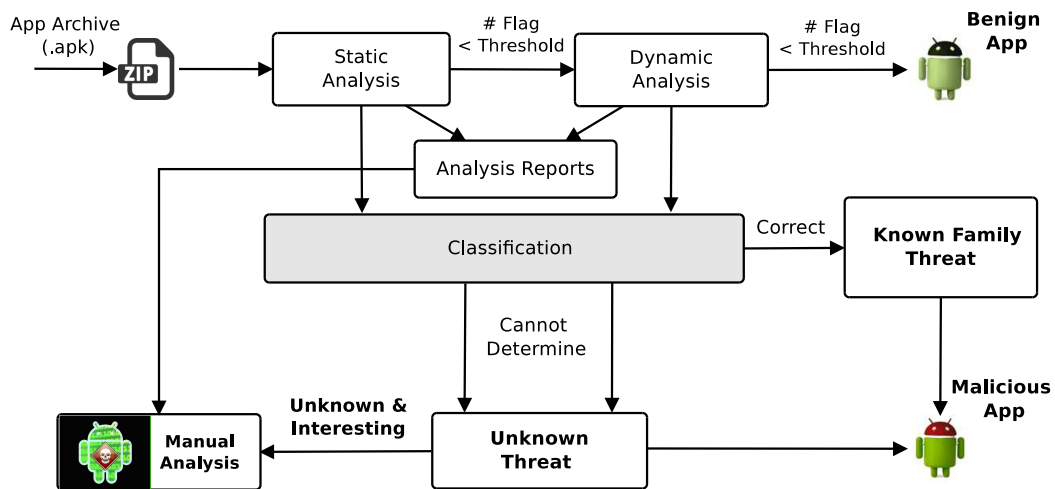

Fig. 16: A Proposed Hybrid Approach for Android Malware Analysis, malware app detection

[3] E. Chin, A. P. Felt, K. Greenwood, D. Wagner, Analyzing Inter-Application Communication in Android, in: Proceedings of the 9th international conference on Mobile systems, applications, and services, MobiSys '11, ACM, New York, NY, USA, 2011, pp. 239–252. doi:10.1145/1999995.2000018.
URL http://doi.acm.org/10.1145/1999995.2000018

[4] Pandaapp, http://www.pandaapp.com/ (Online; Last Accessed 1st March 2014).

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

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

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

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

[9] W. Zhou, Y. Zhou, X. Jiang, P. Ning, Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces, in: Proceedings of the second ACM conference on Data and Application Security and Privacy, CODASPY '12, ACM, New York, NY, USA, 2012, pp. 317–326. doi:10.1145/2133601.2133640.
URL http://doi.acm.org/10.1145/2133601.2133640

[10] ESET - Trends for 2013, http://go.eset.com/us/resources/white-papers/Trends_for_2013_preview.pdf (Online; Last Accessed 11th February).

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

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

[13] F-Secure: Mobile Threat Report Q1 2013, http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q1_2013.pdf (Online; Last Accessed 11th February).

[14] F-Secure: Mobile Threat Report Q3 2013, http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report_Q3_2013.pdf (Online; Last Accessed 11th February).

[15] F-Secure: Mobile Threat Report H1 2013, http://www.f-secure.com/static/doc/labs_global/Research/Threat_Report_H1_2013.pdf (Online; Last Accessed 11th February).

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

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

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

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

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

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

[22] E. Fernandes, B. Crispo, M. Conti, Fm 99.9, radio virus: Exploiting fm radio broadcasts for malware deployment., IEEE Transactions on Information Forensics and Security 8 (6) (2013) 1027–1037.
URL http://dblp.uni-trier.de/db/journals/tifs/tifs8.html#FernandesCC13

[23] R. Fedler, J. Schütte, M. Kulicke, On the Effectiveness of Malware Protection on Android, Tech. rep., Technical report, Fraunhofer AISEC, Berlin (2013).

[24] C. Jarabek, D. Barrera, J. Aycock, ThinAV: Truly lightweight Mobile Cloud-based Anti-malware, in: Proceedings of the 28th Annual Computer Security Applications Conference, ACM, 2012, pp. 209–218.

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

[26] M. Grace, Y. Zhou, Q. Zhang, S. Zou, X. Jiang, RiskRanker: Scalable and Accurate Zero–Day Android Malware Detection, in: Proceedings of the 10th international conference on Mobile systems, applications, and services, MobiSys '12, ACM, New York, NY, USA, 2012, pp. 281–294. doi:10.1145/2307636.2307663.
URL http://doi.acm.org/10.1145/2307636.2307663

[27] G. Suarez-Tangil, J. Tapiador, P. Peris-Lopez, A. Ribagorda, Evolution, detection and analysis of malware for smart devices.

[28] M. La Polla, F. Martinelli, D. Sgandurra, A survey on security for mobile devices, Communications Surveys & Tutorials, IEEE 15 (1) (2013) 446–471.

[29] W. Enck, Defending Users Against Smartphone Apps: Techniques and Future Directions, in: Proceedings of the 7th International Conference on Information Systems Security, ICISS'11, Springer-Verlag, Berlin, Heidelberg, 2011, pp. 49–70. doi:10.1007/978-3-642-25560-1_3.
URL http://dx.doi.org/10.1007/978-3-642-25560-1_3

[30] Android Anatomy and Physiology, http://source.android.com/devices/tech/security (Online; Last Accesed December 25 2013).

[31] Android Kernel Features, http://elinux.org/Android_Kernel_Features (Online; Last Accessed 9th March, 2014).

[32] <permission>, http://developer.android.com/guide/topics/manifest/permission-element.html (Online; Last Accessed 11th February).

[33] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, G. Álvarez, PUMA: Permission Usage to detect Malware in Android, in: International Joint Conference CISIS12-ICEUTE´ 12-SOCO´ 12 Special Sessions, Springer, 2013, pp. 289–298.

[34] Jon Oberhide, DISSECTING THE ANDROID BOUNCER, http://jon.oberheide.org/blog/2012/06/21/dissecting-the-android-bouncer/ (Online;Last Accessed June 1 2012).

[35] Exercising Our Remote Application Removal Feature, http://android-developers.blogspot.in/2010/06/exercising-our-remote-application.html (Online; Last Accessed 11th February).

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

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

[38] M. Conti, B. Crispo, E. Fernandes, Y. Zhauniarovich, Crêpe: A system for enforcing fine-grained context-related policies on android, Information Forensics and Security, IEEE Transactions on 7 (5) (2012) 1426–1438.

[39] M. Nauman, S. Khan, X. Zhang, Apex: extending android permission model and enforcement with user-defined runtime constraints., in: D. Feng, D. A. Basin, P. Liu (Eds.), ASIACCS, ACM, 2010, pp. 328–332.
URL http://dblp.uni-trier.de/db/conf/ccs/asiaccs2010.html#NaumanKZ10

[40] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A.-R. Sadeghi, Xmandroid: A new android evolution to mitigate privilege escalation attacks, Technische Universität Darmstadt, Technical Report TR-2011-04.

[41] M. Ongtang, S. E. McLaughlin, W. Enck, P. D. McDaniel, Semantically rich application-centric security in android., in: ACSAC, IEEE Computer Society, 2009, pp. 340–349.
URL http://dblp.uni-trier.de/db/conf/acsac/acsac2009.html#OngtangMEM09

[42] Android Version History, http://en.wikipedia.org/wiki/Android_version_history (Online; Last Accessed 11th March 2014).

[43] L. Xing, X. Pan, R. Wang, K. Yuan, X. Wang, Upgrading your android, elevating my malware: Privilege escalation through mobile os updating, in: IEEE Symposium on Security and Privacy, 2014.

[44] z4Root, https://github.com/bibanon/android-development-codex/blob/master/General/Rooting/z4root.md (Online; Last Accessed 11th February).

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

[46] CVE, http://cve.mitre.org/ (Online; Last Accessed 11th February).

[47] Android Malware Genome Project, http://www.malgenomeproject.org/ (Online; Last Accessed 11th February).

[48] Z. Yajin, J. Xuxian, Dissecting Android Malware: Characterization and Evolution, in: Proceedings of the 33rd IEEE Symposium on Security and Privacy, Oakland 2012, IEEE, 2012.

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

[50] L. Inc., State of Mobile Security 2012, Tech. rep., Lookout Mobile Security (2012).

[51] C. A. Castillo, Android Malware Past, Present, and Future, Tech. rep., Mobile Working Security Group McAfee (2012).

[52] L. Inc., Current World of Mobile Threats, Tech. rep., Lookout Mobile Security (2013).

[53] C. Lever, M. Antonakakis, B. Reaves, P. Traynor, W. Lee, The Core of the Matter: Analyzing Malicious Traffic in Cellular Carriers, in: Proc. NDSS, Vol. 13, pp. 1–16.

[54] H. T. T. Truong, E. Lagerspetz, P. Nurmi, A. J. Oliner, S. Tarkoma, N. Asokan, S. Bhattacharya, The Company You Keep: Mobile Malware Infection Rates and Inexpensive Risk Indicators, arXiv preprint arXiv:1312.3245.

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

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

[57] G. Andre, P. Ramos, BOXER SMS Trojan, Tech. rep., ESET Latin American Lab (2013).

[58] F. Shahzad, M. A. Akbar, M. Farooq, A Survey on recent advances in malicious applications Analysis and Ddetection techniques for Smartphones.

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

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

[61] avast! Free Mobile Security, http://www.avast.com/free-mobile-security-c?utm_expid=22755838-21.bXJ_mQHnQA6pakUW6PaLQQ.2&utm_referrer=https%3A%2F%2Fwww.google.com%2F (Online; Last Accesed December 25 2013).

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

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

[64] Remote Access Tool Takes Aim with Android APK Binder, http://www.symantec.com/connect/blogs/remote-access-tool-takes-aim-android-apk-binder (Online; Last Accesed December 25 2013).

[65] V. Rastogi, Y. Chen, X. Jiang, Droidchameleon: Evaluating Android anti-malware against Transformation attacks, in: Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, ACM, 2013, pp. 329–334.

[66] M. Zheng, P. P. C. Lee, J. C. S. Lui, ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems, in: DIMVA, 2012, pp. 82–101.

[67] ProGuard, http://proguard.sourceforge.net/ (Online; Last Accessed 11th February).

[68] DexGuard, http://www.saikoa.com/dexguard (Online; Last Accessed 11th February).

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

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

[71] W. Zhou, Y. Zhou, X. Jiang, Hey, You Get Off my Market: Detecting Malicious apps in Official and Third party Android Markets, in: Annual Network and Distributed Security Symposium, NDSS, New York, NY, USA, 2012.

[72] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, Y. Weiss, "andromaly": a behavioral malware detection framework for android devices., J. Intell. Inf. Syst. 38 (1) (2012) 161–190.
URL http://dblp.uni-trier.de/db/journals/jiis/jiis38.html#ShabtaiKEGW12

[73] Y. Feng, S. Anand, I. Dillig, A. Aiken, Apposcopy: Semantics-Based Detection of Android Malware.

[74] P. Faruki, V. Ganmoor, V. Laxmi, M. S. Gaur, A. Bharmal, AndroSimilar: Robust Statistical Feature Signature for Android Malware Detection., in: A. Eli, M. S. Gaur, M. A. Orgun, O. B. Makarevich (Eds.), SIN, ACM, 2013, pp. 152–159.
URL http://dblp.uni-trier.de/db/conf/sin/sin2013.html#FarukiGLGB13

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

[76] L. Lu, Z. Li, Z. Wu, W. Lee, G. Jiang, CHEX: Statically vetting Android apps for Component hijacking vulnerabilities., in: T. Yu, G. Danezis, V. D. Gligor (Eds.), ACM Conference on Computer and Communications Security, ACM, 2012, pp. 229–240.
URL http://dblp.uni-trier.de/db/conf/ccs/ccs2012.html#LuLWLJ12

[77] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, I. Molloy, Android Permissions: a perspective combining risks and benefits, in: Proceedings of the 17th ACM symposium on Access Control Models and Technologies, ACM, 2012, pp. 13–22.

[78] D. Barrera, H. G. Kayacik, P. C. van Oorschot, A. Somayaji, A methodology for Empirical Analysis of Permission-based Security Models

and its Application to Android, in: Proceedings of the 17th ACM conference on Computer and communications security, CCS '10, ACM, New York, NY, USA, 2010, pp. 73–84. doi:10.1145/1866307.1866317. URL http://doi.acm.org/10.1145/1866307.1866317

[79] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, G. Álvarez, PUMA: Permission usage to detect malware in android, in: International Joint Conference CISIS12-ICEUTE´ 12-SOCO´ 12 Special Sessions, Springer, 2013, pp. 289–298.

[80] C.-Y. Huang, Y.-T. Tsai, C.-H. Hsu, Performance Evaluation on Permission-Based Detection for Android Malware, in: Advances in Intelligent Systems and Applications-Volume 2, Springer, 2013, pp. 111–120.

[81] W. Enck, M. Ongtang, P. McDaniel, On Lightweight Mobile Phone Application Certification, in: Proceedings of the 16th ACM conference on Computer and communications security, ACM, 2009, pp. 235–245.

[82] J. Kim, Y. Yoon, K. Yi, J. Shin, S. Center, ScanDal: Static Analyzer for detecting Privacy leaks in Android applications, in: Proceedings of the Workshop on Mobile Security Technologies (MoST), in conjunction with the IEEE Symposium on Security and Privacy, 2012.

[83] H. S. Karlsen, E. R. Wognsen, M. C. Olesen, R. R. Hansen, Study, formalisation, and analysis of dalvik bytecode, in: Informal proceedings of The Seventh Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2012), 2012.

[84] Y. Aafer, W. Du, H. Yin, Droidapiminer: Mining api-level features for robust malware detection in android., in: T. Zia, A. Y. Zomaya, V. Varadharajan, Z. M. Mao (Eds.), SecureComm, Vol. 127 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Springer, 2013, pp. 86–103.
URL http://dblp.uni-trier.de/db/conf/securecomm/securecomm2013.html#AaferDY13

[85] M. Zheng, M. Sun, J. C. S. Lui, DroidAnalytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware, CoRR abs/1302.7212.

[86] JD-GUI, Android Decompiling with JD-GUI, http://java.decompiler.free.fr/?q=jdgui (Online;Last Accessed March 01 2014).

[87] JAD, JAD Java Decompiler, http://varaneckas.com/jad/ (Online;Last Accessed March 01 2014).

[88] H. van Vliet, Mocha, The Java Decompiler, http://www.brouhaha.com/~eric/software/mocha/ (Online;Last Accessed March 01 2014).

[89] SOOT, Soot: a java optimization framework, http://www.sable.mcgill.ca/soot/ (Online; Accessed March 01 2014).

[90] WALA, T.j. watson libraries for analysis (wala), http://wala.sourceforge.net/wiki/index.php/ (Online; Accessed March 01 2014).

[91] H. Inc., Fortify static code analyzer, http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812 (Online; Accessed March 01 2014).

[92] E. William, O. Damien, M. Patrick, C. Swarat, A Study of Android Application Security, in: USENIX Security '11, USENIX, San Francisco, ca, 2011.

[93] D. Octeau, S. Jha, P. McDaniel, Retargeting Android applications to Java bytecode, in: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, ACM, 2012, p. 6.

[94] A. Bartel, J. Klein, Y. Le Traon, M. Monperrus, Dexpler: converting android dalvik bytecode to jimple for static analysis with soot, in: Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, ACM, 2012, pp. 27–38.

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

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

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

[98] A. Reina, A. Fattori, L. Cavallaro, A System call-centric analysis and stimulation technique to automatically reconstruct Android Malware behaviors, EUROSEC, Prague, Czech Republic.

[99] E. William, G. Peter, C. Byunggon, C. Landon, TaintDroid : An Information Flow Tracking System for Realtime Privacy monitoring on Smartphones, in: USENIX Symposium on Operating Systems Design and Implementation, USENIX, 2011.

[100] I. Burguera, U. Zurutuza, S. Nadjm-Tehrani, Crowdroid: Behavior-based Malware Detection System for Android, in: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices, ACM, 2011, pp. 15–26.

[101] L. K. Yan, H. Yin, Droidscope: Seamlessly reconstructing the OS and Dalvik Semantic views for Dynamic Android Malware Analysis, in: Proceedings of the 21st USENIX Security Symposium, 2012.

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

[103] DARE: Dalvik Retargeting, http://siis.cse.psu.edu/dare/ (Online; Last Accessed 11th February 2013).

[104] Dedexer, http://dedexer.sourceforge.net/ (Online; Last Accessed 11th February 2013).

[105] JEB Decompiler, http://www.android-decompiler.com/ (Online; Last Accessed 11th February 2013).

[106] Similarities for Fun & Profit.

[107] V. Roussev, Data Fingerprinting with Similarity Hashes, Advances in Digital Forensics.

[108] V. Roussev, Building a better similarity trap with statistically improbable features, in: System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on, IEEE, 2009, pp. 1–10.

[109] Andrubis (2012).
URL http://anubis.iseclab.org/

[110] A. Desnos, P. Lantz, Droidbox: An android application sandbox for dynamic analysis (2011).
URL https://code.google.com/p/droidbox/

[111] APKInspector (2013).
URL https://github.com/honeynet/apkinspector/

[112] ded: Decompiling Android Applications, http://siis.cse.psu.edu/ded/ (Online; Last Accessed 11th February).

[113] R. Xu, H. Saïdi, R. Anderson, Aurasium: Practical policy Enforcement for Android applications, in: Proceedings of the 21st USENIX conference on Security symposium, USENIX Association, 2012, pp. 27–27.

[114] Google Bouncer: Bad guys may have an app for that, http://www.techrepublic.com/blog/it-security/google-bouncer-bad-guys-may-have-an-app-for-that/7422/ (February 2012).

[115] CopperDroid, http://copperdroid.isg.rhul.ac.uk/copperdroid/index.php (February 2012).

[116] J. Kornblum, Identifying almost Identical Files using Context Triggered Piecewise Hashing, Digital Investigation 3 (2006) 91–97. doi:10.1016/j.diin.2006.06.015.
URL http://dx.doi.org/10.1016/j.diin.2006.06.015

[117] Drozer - A Comprehensive Security and Attack Framework for Android, https://www.mwrinfosecurity.com/products/drozer/ (Online; Last Accessed 11th February 2013).