A

Ph.D. Thesis

on

Techniques for Analysis and Detection of Android Malware

submitted in partial fulfillment for the degree of

Doctor of Philosophy

(Computer Science and Engineering)

in

Department of Computer Science and Engineering

(March 2016)

Supervisor(s):
Dr. Vijay Laxmi
Prof. Manoj Singh Gaur

Submitted By:
Parvez Faruki
(2012RCP9518)

MALAVIYA NATIONAL INSTITUTE OF TECHNOLOGY, JAIPUR

Department of Computer Science and Engineering

Malaviya National Institute of Technology , -

Certificate

We hereby certify that the Thesis titled, "**Techniques for Analysis and Detection of Android Malware**" submitted by **Parvez Faruki (2012RCP9518)** is the research work done under our supervision, thus is accepted and finalized for submission in partial fulfillment of Ph.D. Program.

Place:	**Supervisor(s):**
	Prof. Manoj Singh Gaur, (Professor)
Date:	Dr. Vijay Laxmi, (Associate Professor)

Declaration

I, Parvez Faruki, declare that I own the research work introduced in this Thesis titled, “Techniques for Analysis and Detection of Android Malware” and the research contents used in this thesis. I with this assure that:

- The research work produced in this thesis is for the partial fulfillment of the degree of “*Doctor of Philosophy*” at MNIT, Jaipur.
- I have stated all the major resources used for the help.
- Where I have used proper citation for the work proposed by other researchers and quoted the source. This entire thesis belongs to me with the some exception of such citations.
- Where I have taken references of previously published work of other researchers and this is always clearly attributed.
- I have clearly stated any part of this Thesis that has been previously submitted for a degree or any other qualification at MNIT or any other institution.

Signed: ______________________________

Date: ______________________________

<u>Acknowledgement</u>

It is my privilege to have worked under the supervision of Dr. Vijay Laxmi and Prof. Manoj Singh Gaur during my M.Tech and PhD at the department of Computer Science and Engineering, Malaviya National Institute of Technology, Jaipur. Words are not enough to express my gratitute for Dr. Vijay Laxmi and Dr. Manoj Singh Gaur whose guidance helped me improve. I thank my supervisors for the support and guidance that has and will motivate me to work hard.

I would like to express my deep appreciation to a lot of people for making my PhD, a special one. I thank Dr. Mauro Conti, (Associate Professor, University of Padova, Italy) for being very critical about the submitted research. Heart felt gratitude to Dr. Muttukrishnan Rajarajan, (Professor, City University London) for his inputs in the Android Platform Security research.

I thank dear friend Manoj Bohra, PhD scholar, computer science and engineering for all the help during my PhD tenure. I thank Ammar Bharmal and Vijay Ganmoor for the company and tireless efforts in our research at CSE MNIT. Thank you Vinesh for being such a nice friend. I thank Lovely, Shweta, Jitendra, Rishabh, Pallavi and Utkarsh for teaming up with me in Android Platform Security research at MNIT.

I thanks my DREC members Dr. Girdhari Singh (MNIT, Jaipur), Dr. Dinesh Gopalani (MNIT, Jaipur) and Dr. Preeti Singh (LNMIIT, Jaipur) for being very kind and encouraging.

I thank my family for supporting me. Finally, I thank Allah for providing the strength to work hard during my PhD tenure.

Dedications

*I dedicate my PhD Thesis to my father **Kaiserkhan**, wife **Afsana** and mother **Hasinaben** without whom, I would not have been able to pursue my doctoral research with utmost dedication. I thank my kids Alisha, Maheer and Zaid for the motivation and support during my studies at MNIT.*

Abstract

The spread and acceptance of mobile devices, specifically smartphone and tablets have exploded since launch of Google Android. The Internet enabled mobile devices offer rich services like location tracking, wearable computing, multimedia services and a comprehensive Application Programming Interface (API) framework. The exponential increase of Android devices has attracted the attention of malware authors to leverage monetary benefits by targeting the Android OS. The malicious software evades mobile device security once it gains access to sensitive resources.

Malware authors use obfuscation and targeted infection techniques to infect the users. The traditional computing resources have mature detection capabilities. However, their direct adaptation on mobile devices is a challenge on account of limited processing capability, limited memory and battery constraints. These issues make the replication of Personal Computer (PC) based detection methods infeasible for mobile Operating System (OS). Furthermore, cloud-based detection scenarios have their privacy invasion concerns. In this Thesis, we propose, design and develop multiple static and dynamic analysis and detection techniques for Android malware. A single analysis technique can be evaded by a targeted, adversarial malware. The use of multiple analysis and detection techniques improves the code coverage.

In the first step, we propose ApPRaISe, Android permission-based $n-Set$ analysis technique to identify the sequence of dangerous permissions for classifying the malicious apps. In particular, we extract the Androidmanifest permissions and map the defined permissions to its use in the Dalvik bytecode to identify the over-privileged apps. The proposed methodology evaluates 8,341 benign Google Play and 6,298 malicious apps with a reasonable accuracy. However, it was observed that the permission based malicious app detection incurs high false positives.

Next, we developed AndroSimilar, a robust byte based statistical feature signature to detect repackaged malware. The proposed approach generates a variable size signature to identify repackaged malicious apps. The fixed-size byte sequence is based on empirical probability of entropy computed using sliding window. The values are computed in a sliding window fashion. The popular features are identified according to their neighborhood rarity. We extend the AndroSimilar algorithm for malware family detection to reduce the number of signatures. We cluster the malware variants with high similarity and generate a variant signature discarding the common features.

The evolving malware threats circumvent the signature-based techniques. To review the importance of advanced and targeted infection capabilities such as obfuscation and repackaging, we developed an Android bytecode obfuscator "DroidsHornet". We analyze the impact of trivial code obfuscation techniques on our proposed solution AndroSimilar. In fact, the proposed AndroSimilar identified the malicious apps from Google Play that remained undetected from the top commercial anti-malware.

Second, we propose detection of malicious apps that perform covert operations such as sending SMS, making voice calls, recording audio/video, clicking pictures without the user knowledge or consent. We identify such *covert actions* as *sensitive feature misuse*. Our proposal *CONFIDA* is an inter-component-communication based malware analysis technique to detect the sensitive feature misuse employed by evolving Android malware. We generate a precise Dalvik bytecode Inter-Component Communication (ICC) based Component Interaction Graph (CIG) considering the asynchronous nature of ICC API. Furthermore, we perform reverse reachability analysis to identify if the *feature usage* or *behavior* is initiated with legitimate user interaction or hidden malicious behavior.

The techniques like environment detection, reflection and dynamic code loading evades the static analysis. The malware identifies analysis system and evades the analyzer with a benign behavior. The existing machine learning classification techniques select malicious apps randomly without considering their arrival pattern, downgrading the classification performance during real time detection. We overcome the limitations of static analysis with proposed dynamic analysis Sandbox to identify runtime information and detect environment-aware malware.

The proposed Sandbox modifies the existing static properties of the default emulator and resembles a real Android device to reveal the hidden malicious behavior. We emphasize the importance of timeline in malware dataset selection to underline its influence on the machine learning-based malware classification techniques. We have experimentally evaluated 6,743 Google Play and 2,786 malicious apps with classification accuracy better than the existing approaches reported in the literature. The multiple proactive analysis is useful in early detection and defense against the emerging malware threats.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Android smartphone OS has captured nearly 2/3 smartphone market, leaving its competitors iPhone Operating System (iOS), Windows Mobile OS and Blackberry far behind [8, 9]. The smartphones have been prevalent in the previous decade. However, the launch of Android and iOS generated enormous attraction among the users and developers worldwide. Smartphones have become ubiquitous due to broad connectivity options such as Global System for Mobile communications (GSM), Code Division Multiple Access (CDMA), Wireless Fidelity (Wi-Fi), Global Positioning System (GPS), Bluetooth and Near Field Communication (NFC). The Gartner '15 smartphone sale reports 42.3% increase of Android devices compared to previous year [9]. The Android device market share increased from 66% to 78%, a substantial rise of 12% [8]. The nearest Android competitor iOS share declined 4% from 19 to 15 percent [8]. The ubiquitous Internet connectivity, a wealth of personal information (contacts, messages, social network access, browsing history and banking credentials) has attracted malware developers to target Android platform. The Android premium-rate SMS Trojans, spyware, botnets, aggressive adware has also spread through Google Play [2, 10, 11].

Mobile Malware: According to [12], malware an acronym for malicious software is "deliberately fulfilling the harmful intent of an attacker". The personal computer correlates malware, according to its functionality such as Virus, Worm or Trojan. The initial days of malware development were motivated to demonstrate problems within the system to earn respect. The systems are shifting to online computing, web-based transactions, banking and online bill payments; hence, the prime focus is monetary gain [13]. The Android malware growth has turned exponential due to the availability of obfuscation and protection tools [14]. In [15], the authors predicted similar trends.

1. Smartphone growth is phenomenal as they have become accessible, powerful, cheap and easy to use with the growth of wireless networks. The introduction of Android, an open source mobile OS has fuelled the growth.

2. Google Android has a liberal app vetting procedure. The malware authors misuse the facility and infect online app stores to propagate malicious apps [4, 10, 16].

3. A Smartphone is strictly personal device consisting sensitive information such as Phone numbers, SMS, Payment information, Login credentials, and a dearth of personal information unlikely on a PC. Moreover, the Mobile Internet Devices (MID) are exposed to various attacks due to the large number of facilities available on a single device [15, 17].

Google Play, the official Android app market hosts third-party developer apps for a nominal fee. Google Play hosts more than 1.5 million apps with a large number of downloads each day [18]. Unlike Apple, Google Play does not vet the uploaded apps manually. The official market employs Bouncer [19, 20], a dynamic analysis engine. The Bouncer protects Google Play against the malware; however, it does not perform vulnerability analysis [21]. Malware authors take advantage and exploit such vulnerable apps and divulge the private user information to inadvertently harm the app store and developer reputation. Moreover, Android open source philosophy permits the installation of third-party market apps, stirring up dozens of regional and international app-stores [22, 23]. However, the adequate protection methods and app quality at the third-party app store is a concern [11].

Exponential increase in malicious apps have forced the anti-malware industry to carve out robust and efficient methods suited for on-device analysis in spite of the existing constraints. The existing commercial anti-malware solutions employ signature-based detection due to its implementation efficiency and simplicity [24]. The signature-based methods can be evaded by code obfuscation; necessitating a new signature for each malware variant [25]; Thus, it coerces the anti-malware client to update the signature database at regular intervals. Furthermore, the mobile OS resource constraint mandates the use of distributed computing [26, 27]. The exponential increase of malware variants necessitates automated analysis methods.

Off-device malware analysis methods are needed to understand the malware functionality. The samples can be analyzed manually to extract the malware signatures. Automated analysis helps the malware analyst generate a timely response for detecting malicious application. Static analysis can quickly and precisely identify malware patterns. Still, static analysis is evaded by code transformations and Java reflections [28]. Thus, dynamic analysis approach can be employed to extract stealthy malicious behavior. The researchers have proposed solutions to analyze and detect the Android malware threats. Some of these are even available as open-source. These solutions can be characterized using the following three parameters:

1. *Goal* of the proposed solution can be either app security assessment, analysis or malware detection. The app security assessment solutions determine the vulnerabilities, which if exploited by an adversary, harms the user and device security. The analysis techniques verify the malicious behavior; however, detection solutions aim to prevent the on-device installation.

2. *Methodology* to achieve the above goals can be either *static* or *dynamic* analysis. Control-flow and data-flow analysis are examples of formal static analysis [27]. In *dynamic* analysis, apps are executed/emulated in a sandboxed environment, to monitor the activities and identify anomalous behaviors that are difficult with static analysis.

3. *Deployment* of the above discussed solutions.

1.1 Objectives

The purpose of this Thesis is to study analysis tools, techniques and develop multiple static and dynamic analysis techniques to analyze and detect "single malicious app" with improved analysis coverage.

To achieve the goal, we focus the following objectives:

1. Review the malware growth and present state of malware on Android Platform. Identify the available analysis techniques for detection of Android malware.
2. Propose, develop and implement analysis and detection techniques to improve the existing static and dynamic analysis for malware detection from app markets, emphasizing automated analysis and detection techniques.
3. Identify the complementary use of static and dynamic analysis techniques to detect evolving malicious apps from a large number of ever increasing malware. The sophisticated malware includes repackaged malware, covert malicious behavior executed without user knowledge, analysis environment-aware malware to name a few.

1.1.1 Thesis Impact

The proposed analysis techniques presented in this Thesis will improve the defense against individual Android malware detection. Furthermore, the proposed research will aid the third party developer and device user analyze the potential app. Finally, the multiple analysis techniques can be used to generate analysis reports for app vetting within an organization.

1.2 Motivation and Contributions

In this Thesis we have identified issues like ever increasing malicious apps, growing use of obfuscation, code protection and sophisticated evasion techniques targeting the Android devices. Based on extensive literature review of the challenges in malware analysis, following are the motivations and contributions of this Thesis:

1. The motivation of the "ApPRaISe" model is derived from the fact that Android Security depends on mandatory permission-based mechanism to protect sensitive resources. The existing state-of-the-art considered the individual permissions to detect malicious apps. We propose an $n-Set$ Permission model identify the app risk. We employ Android Permission Risk Model to filter the applications that do not extensively use $n-Set$ dangerous permissions. Furthermore, we extended the $n-Set$ permissions model to map the

declared Permissions with their use in Dalvik bytecode. The second part identifies the over-privileged APK files.

2. The repackaged Android apps pollute the application markets. It is difficult to identify the repackaged applications based on the permission analysis. Jiang et al. [29] report 86% repackaged malware from the Android malware GENOME [29] dataset. Hence, we propose "AndroSimilar", a robust byte-based statistical signature to detect the repackaged applications. The proposed approach generates a variable size signature to identify repackaged malicious apps. The fixed-size byte sequence is based on empirical probability of entropy computed using sliding window.

3. We extend the aforesaid "AndroSimilar" considering the ever increasing malicious applications. We cluster the malware variants with high similarity and generate a variant signature discarding the common features. The proposed robust signature is effective against unseen variants of known malware.

4. Since the ICC forms the core of Android application development model we propose a novel static analysis approach CONFIDA to identify the covert-malicious behavior employed by advanced Android malware. The proposed CONFIDA generates CIG considering the ICC and data flow analysis. The proposed approach can detect advanced malware evading the important user-interaction, a necessity to invoke sensitive functionality such as sending SMS, Phone call, or audio/video recording without user knowledge.

5. The evolving Android malware (e.g., Anserver, FakeNetFlix, FakeFacebook) employ code obfuscation, repackaging and anti-analysis techniques to evade the anti-malware. To analyze the impact of obfuscation techniques, we developed a prototype implementing popular `x86` Transformation techniques [30] and generated unseen malware. In particular, we evaluated the anti-malware techniques against our proposal AndroSimilar signature and Androguard, a robust semantic based static-analysis tool. Our proposal AndroSimilar performs better than the existing anti-malware techniques. Furthermore, we discuss the limitations of AndroSimilar when evaded by a particular code-transformation.

6. The evolving malicious apps [31] have inbuilt capability to identify the analysis based emulated environment. Once the app identifies the analysis sandbox or virtual environment, it behaves benign hiding the malicious function-

ality. To uncover such environment aware malware, we propose a framework that makes a malware believe that it is being executed on the real Android device instead of the emulator, an alibi for development or an analysis system. We target the Android system features with modified static emulator properties and enrich the virtual device with essential user information. To explore the execution paths, we integrate user input simulation with *intent* broadcasts.

In particular, we propose the detection of analysis aware Android malware within a modified Sandbox to reveal the hidden behavior. We argue that random malware sample selection strategies employed by the existing techniques deteriorates the detection performance.

In this Thesis, our focus is to propose, design and develop multiple static and dynamic analysis techniques for Android malware detection. We believe that a single method can be evaded by a targeted, adversarial malware. However, multiple analysis methods improving the detection rate.

1.3 Thesis Organization

The remainder of this Thesis is organized as follows. Chapter 2 evaluates the existing work on Android security, issues and malware penetration with an emphasis on single malware app detection. In Chapter 3, we propose a permission based $n-Set$ analysis technique to differentiate malicious and potential risky apps from a large set of submitted applications. Chapter 4 presents a robust statistical signature to detect repackaged applications and variants of Android malware plaguaging the Android distribution system. Chapter 5 explores CONFIDA, our proposed Component Interaction Graph technique to detect covert malicious behavior. In Chapter 6 we evaluate the impact of code transformation techniques against existing static analysis tools. To leverage the dynamic analysis, Chapter 7 describes the design and implementation of a dynamic analysis technique to detect analysis environment aware apps. Finally, we conclude the Thesis with pointers to the future directions.

Chapter 2

A Review of Android Malware Analysis

Android devices have gained enormous market share due to the open architecture and its popularity among users and third party developers. The increased popularity of the Android devices and associated monetary benefits have attracted the attention of malware developers, resulting in a tremendous rise of the Android malware from 49 families in 2010 to 273 families, an increase of 676% unique malicious instances in 2014 [6, 7, 32].

In this chapter, we discuss the Android security enforcement mechanisms, malware threats and related issues for applications popularly called apps. We will interchangeably use apps for application in this Thesis. The chapter details an insight into the strength and shortcomings of the known research methodologies and provides a base towards proposing novel malware analysis and detection techniques.

2.1 Android App and Security Architecture

Google developed Android under Android Open Source Project (AOSP) and is promoted by the Open Handset Alliance (OHA). The OHA is a consortium of Original Equipment Manufacturer (OEM), chip-makers, carriers and application developers. The Android apps are developed in Java; however, the native code and shared libraries are developed in C/C++. The Android OS architecture is illustrated in Figure 2.1. The bottom-most layer, the Linux kernel is customized for the embedded environment. Android is developed on top of Linux kernel due to its robust driver model, efficient memory & process management and networking support. Currently, Android supports two instruction set architectures:

1. Advanced RISK Machines (ARM), prevalent on Smartphone and Tablets.
2. `x86`, more frequently used among the MID.

As illustrated in Figure 2.1, at the top of the kernel, resides shared native code and libraries developed in C/C++.

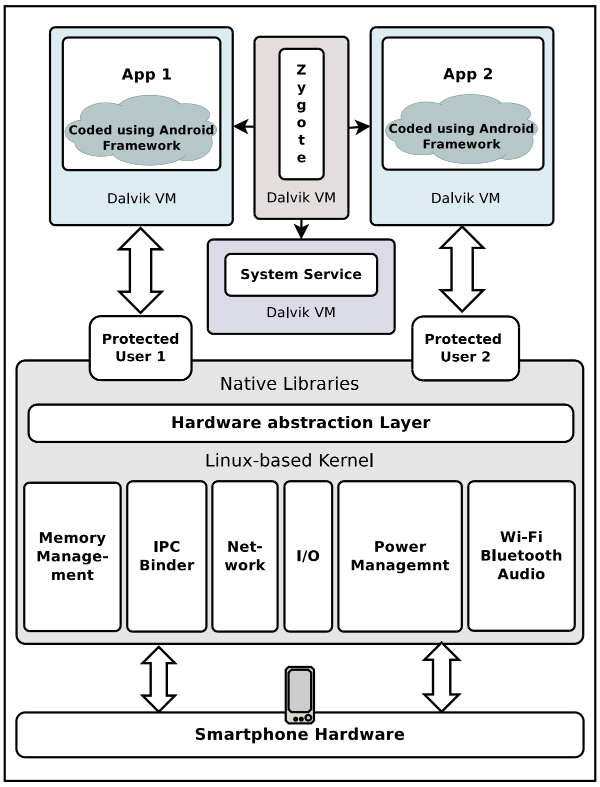


Figure 2.1: Android architecture envisaged by Google [1].

Android app code developed in Java language is converted to Dalvik bytecode which is executed on the *Dalvik Virtual Machine (DVM)* as illustrated in Figure 2.1. The Virtual machine based architecture provides process isolation, an important security feature of Android platform. Once the OS boot completes, the first parent process known as *zygote* initializes the Dalvik VM by pre-loading the core libraries. The *zygote* loads the newly forked processes to speed up the app loading. Finally, the application framework provides a uniform and concise view of the Java libraries to the app developers. Android protects the sensitive functionality such as telephony, GPS, network, power management, radio and media as system services with the permission-based model.

2.2 Android APK

The `Android PacKage File (APK)` is a zip archive illustrated in Figure 2.2. In particular, the `AndroidManifest.xml` stores meta-data such as: (1) Package name; (2) Resource permissions; (3) Components; (4) version support; and (5) shared libraries. `res` stores icons, images, string/numeric/color constants, UI layouts, menus, animations compiled into a binary. `assets` stores the non-compiled resources. Executable file `classes.dex` stores the Dalvik bytecode to be executed inside the Virtual Machine. `META-INF` stores the signature of the app developer certificate to verify the third party developer identity.

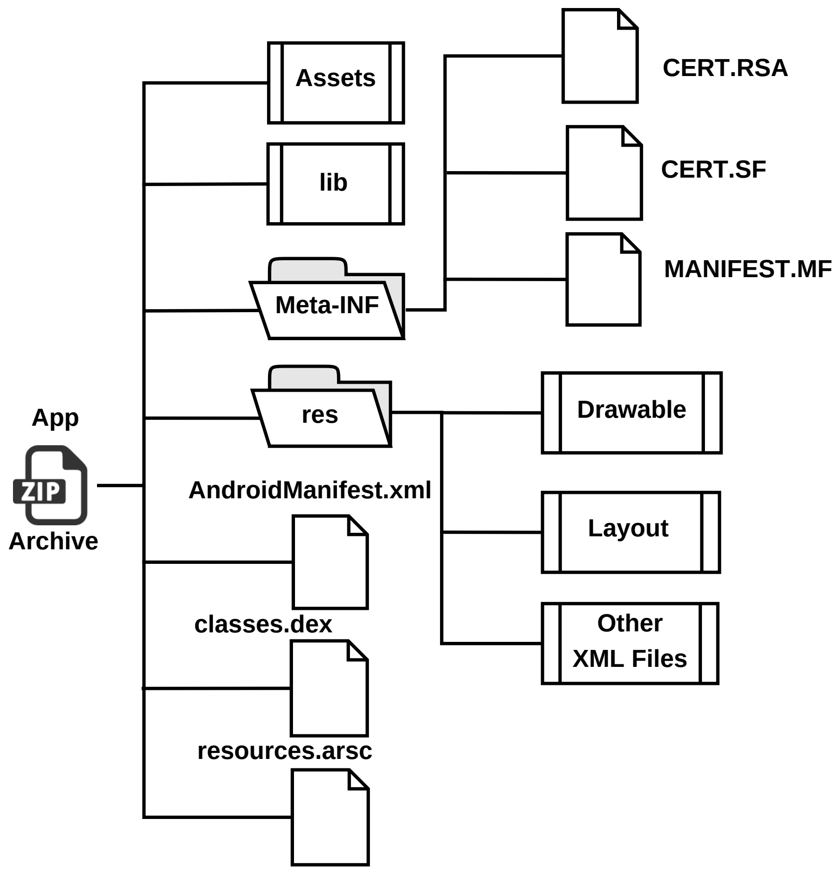


Figure 2.2: Android PacKage (APK) structure.

During the application development, Java code is compiled to generate corresponding `.class` files. *dx* tool merges the `.class` files into a single *Dalvik EXecutable (DEX)*. The *.dex* stores executable Dalvik bytecode to be executed on the DVM. The app archive installation is based on permissions defined in the APK. A user must approve all permissions to install an app. The Android platform does not support individual dangerous permission revocation till Android Lollipop 5.0. The approved permissions cannot be revoked once the app is installed on device. The Android architecture is aware of the permission type and its use requested by the APK developer. A user must be aware of the implications of dangerous permissions, and its actual use by the app requesting a permission.

2.2.1 App Components

An Android app is built from four basic components discussed below. Component is made accessible to the other apps by explicitly exporting it for code reuse by the same developers. The declared component(s) can be invoked or executed independently since the app component communication is asynchronous. Android app has multiple entry-points depending on the number of components an application defines.

- *Activity*: It is the user interface component of an app. Any number of activities can be declared within the manifest depending on the developer requirements. Apart from some pre-defined task, an activity can also return the result to its caller. Activity(s) can be launched using the *Intent.*[1].
- *Service*: A Service component is used to run a background task. The long running tasks such a playing music, data download, updating an application or uploading videos can be achieved using the Service. The Service component does not need any User Interface (UI). Service normally gives notifications to the user. The Service component can be launched by: (i) Attaching Service to a particular Activity. Here, the Service ends as soon as an Activity stops. (ii) Run the Service independently of any app. In this case, the Service component keeps running even if the app is stopped.

[1]*Intent:* The Android Intent is a conceptual description of an operation to be performed. Intent provides a facility for performing late runtime binding between the code between different apps using high-level abstraction for Inter-Process Communication (IPC), internally handled by the Binder IPC. Intent is used to launch components like activity, service and system broadcast.

- *Broadcast Receiver*: This component listens to the Android system-generated events of the device. For example, `BOOT_COMPLETED`, `SMS_RECEIVED` is a system event. Other apps can broadcast their application-defined events, which can be handled by other apps using the Service component. A Broadcast Receiver can be used to develop apps such as Caller Number finder or a Short Message Service (SMS) blocker

- *Content Provider*: Content provider, also known as the *data-store* provides a consistent interface for data access within and among different apps. The Content Provider allows data use. The default Android `Contact` app is an example of the Content Provider. The device contacts can be made available to multiple apps like the SMS app and Phone Dialer. Data store is accessible through the app-defined Uniform Resource Identifier (URI). The Content Provider provides encapsulated data access.

A Component is made accessible to the other apps by explicit export to the manifest file. However, the exportting is static, which cannot be changed at runtime. A Component can be protected explicitly by manifest permission. An unprotected Component becomes vulnerable to the misuse by malicious apps. Furthermore, inter-app component misuse attacks can be launched to evade anti-malware still employing “single” malicious app analysis and detection techniques.

2.2.2 Inter-Component Communication (ICC)

The Android Security protects apps and data using a combination of system-level and ICC [33]. App components interact with each other at a high-level abstraction of IPC using *Intent*, handled by the Binder IPC driver. An app runs with a unique `user-id` within the Android Sandbox. The Android middleware mediates the ICC between application and components. The access to components is restricted with a ‘permission label’. When a component initiates ICC, the reference monitor looks at the permission labels assigned to its container app. If the target component access permission label is in the collection, it allows ICC to be initiated. If the label does not belong to the Group IDentifier (GID), ICC establishment is refused even if the component is declared within the same app. The app developer define security policy, whereas assigning permission(s) to the components in an application specifies an access policy to protect its resources.

The Apps invoke *activities*, *services* and send the broadcast events using the Intents that refer to an explicit address of the receiver components using class/package name field. Depending upon the type of action, category and data fields, the system sends implicit Intents to one or more matching receiver components. A Component registers itself to receive the Intent(s) using one or more *intent-filter* defined in the Android manifest. It is also specified if the intent can accept the kind of action category.

2.3 Android Malware Threat Perception

Figure 2.3 illustrates the timeline of some notable malware families of Android during 2010-2013. The premium rate SMS Trojans have successfully penetrated the Google Play by evading the Bouncer, the official market anti-malware [29]. A large number of malware apps have exploited root-based attacks such as *rage-against-the-cage* [34], *gingerbreak* [35] and *z4root* [36] to gain superuser privileges and control the device. Another notable exploit *master-key* attack [37] has successfully evaded anti-malware on Android devices from OS versions 1.6 to 4.4.

Each quarter, the commercial anti-malware reports exponential increase in the new families and existing malware variants [38, 10]. Lookout Inc. reports global malware infection rate likelihood as 2.61% [16]. In [39] the authors, used smartphone Domain Name Resolution (DNS) traffic from United States and reported 0.0009% infection. Furthermore, Truong et al. [40] instrumented Carat app [41] and report 0.26% and 0.28% for McAfee and Mobile-Sandbox malware dataset respectively. The present Android threat perception and malware infection rate report significant variation with the commercial anti-malware studies.

2.4 Malware Survival Techniques

The malware analysis is either static or dynamic. The static analysis techniques parse the source code without executing the app to identify malicious behavior. Static analysis covers all execution paths of an app, which is valuable in security analysis. However, static analysis techniques can be evaded by obfuscation, protection and encryption methods. The dynamic analysis is a black box methodology that identifies the traces of executed app with the system. Dynamic analysis

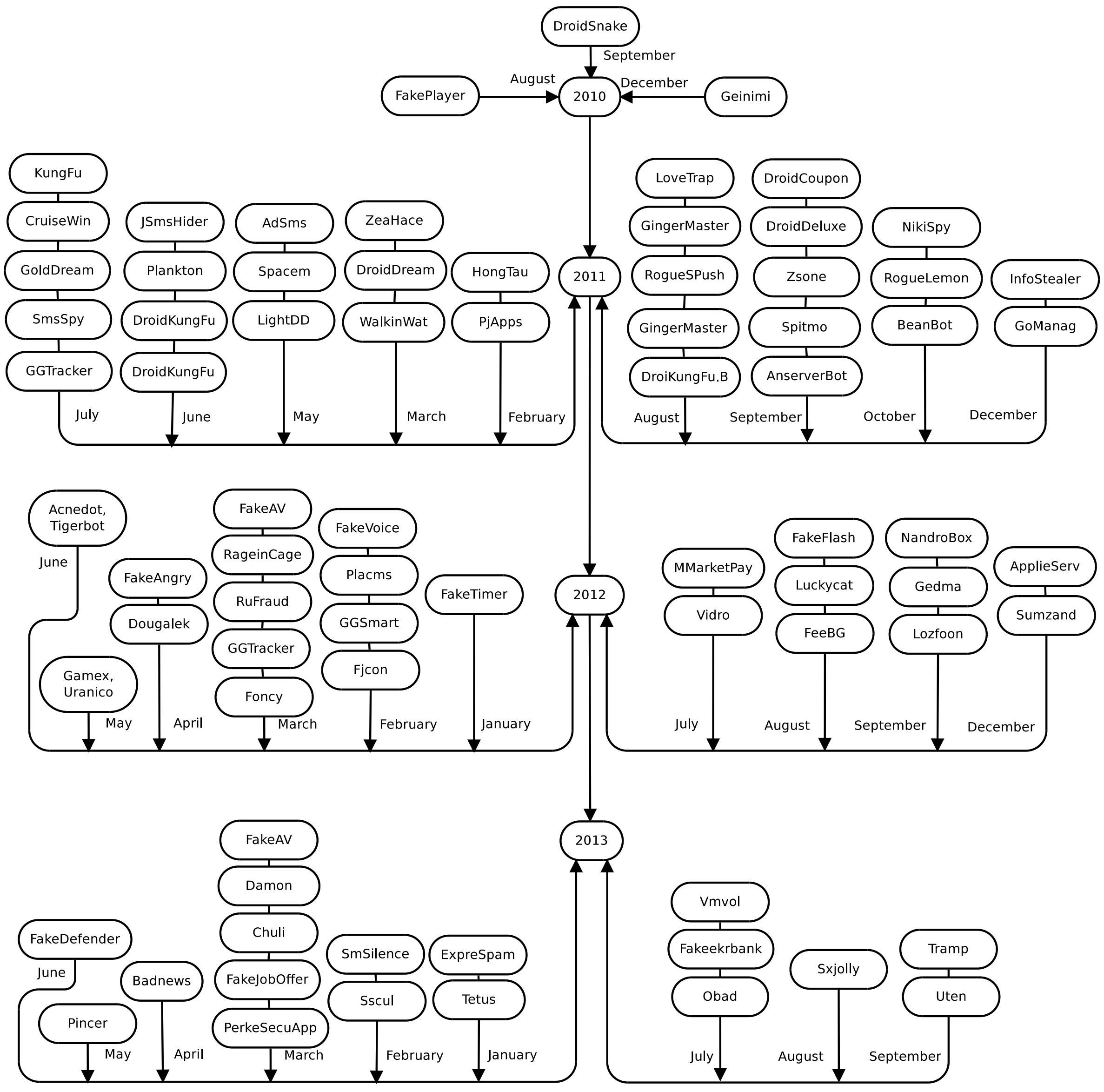


Figure 2.3: Android malware chronology [2, 3, 4, 5, 6, 7]

techniques are effective against obfuscated and encrypted malware. The malware authors have recently adopted Dalvik bytecode transformations and analysis environment detection techniques to evade the dynamic analysis based anti-malware. In this section, we summarize the popular Android malware survival and stealth techniques to evade existing analysis.

2.4.1 Repackaging Popular Apps

Repackaging is a process of reverse engineering a popular app from the app markets. The malicious payload is added to rebuild the APK. The trojan app is disseminated from the less monitored local app markets. The repackaging procedure is illustrated in Figure 2.4. The following section discusses the main steps involved in app repackaging:

1. Download the popular free/paid app from the popular app store(s).
2. Obtain the assembly code of the APK with *apktool* disassembler [42].
3. Generate a malicious payload, convert it in Dalvik bytecode with the *dx* [43] tool.
4. Add the malware payload into a benign app. Modify the `AndroidManifest.xml` and/or *resources* if required.
5. Assemble modified source again using *apktool*.
6. Distribute the repackaged app as a free app or free version of popular app from less monitored third party markets.

Repackaging is one of the most common malware app generation technique. More than 80% samples from the Malware Genome Dataset have repackaged malware variants [11] of the legitimate official market apps. Repackaging can be used to generate large number of malware variants. It can also be used to spawn unseen variants of the already known malware. As the signature of each malware is unique, the commercial anti-malware can be easily evaded by the unseen malware. Repackaging is a big threat as it can pollute the app market and hurt the developer reputation. The malware authors divert the revenues by replacing their own advertisements in place of the developer advertisements.

AndroRAT APK Binder [44] repackages and generates a trojanized version of a popular and legitimate APK adding remote access functionality. The adversary can remotely force the infected device to send SMS messages, setup voice calls, reveal the device location, record audio, video and access the device files.

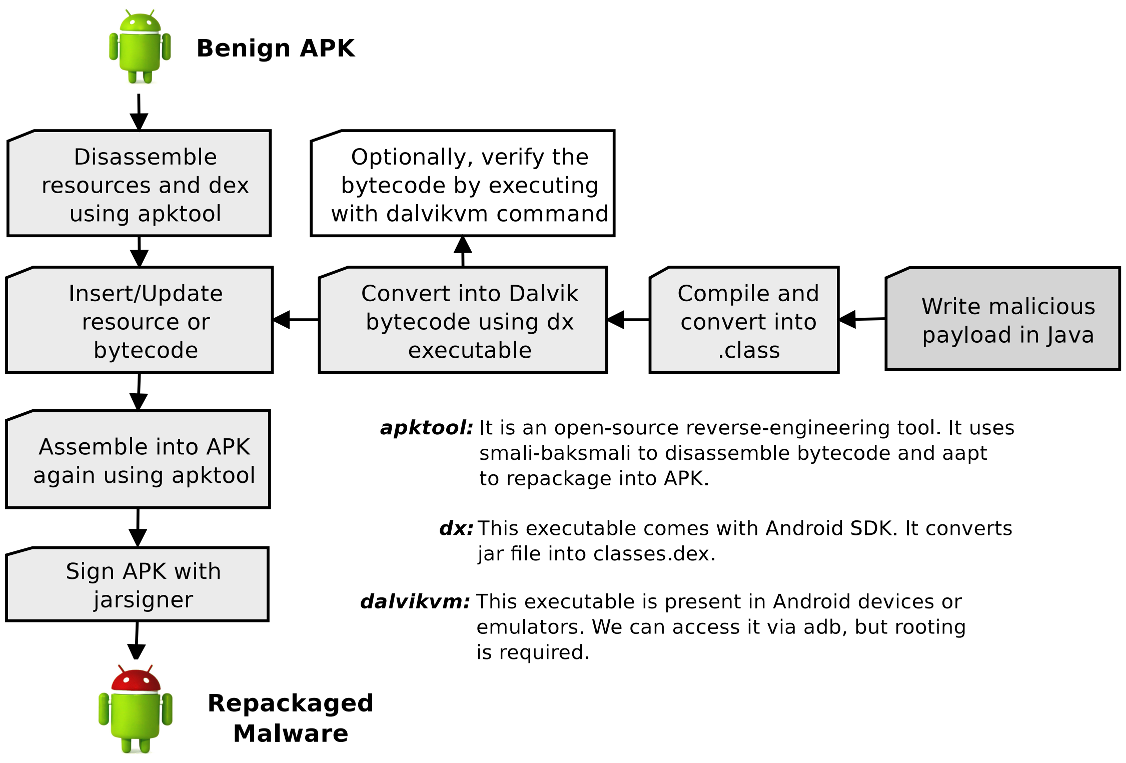

Benign APK
Disassemble resources and dex using apktool
Insert/Update resource or bytecode
Assemble into APK again using apktool
Sign APK with jarsigner
Repackaged Malware
Optionally, verify the bytecode by executing with dalvikvm command
Convert into Dalvik bytecode using dx executable
Compile and convert into .class
Write malicious payload in Java
apktool: It is an open-source reverse-engineering tool. It uses smali-baksmali to disassemble bytecode and aapt to repackage into APK.
dx: This executable comes with Android SDK. It converts jar file into classes.dex.
dalvikvm: This executable is present in Android devices or emulators. We can access it via adb, but rooting is required.

Figure 2.4: App repackaging.

2.4.2 Drive-by Download

An attacker can employ social engineering, aggressive advertisements having malicious URL to force or lure the user to download malware. Optionally, a drive-by download may disguise as a legitimate application and coax the user to install the malicious app. *Android/NotCompatible* [45] is a well known drive-by-download app.

2.4.3 Dynamic Payload

An app can also embed malicious payload as an executable `apk/jar` in encrypted or plain format inside the APK resources. Once installed, the app decrypts the payload. If the payload is a `jar` file, malware loads `DexClassLoader` API and execute dynamic code. The user unknowingly installs the embedded `apk` disguised as important update. The app can execute native binaries using `Runtime.exec` API, an equivalent of Linux `fork()/exec()`. *BaseBridge* [29] and *Anserverbot* [29] malware employ the above discussed technique. Some malware families do not embed malicious payload as a resource, but rather download them from the remote server and successfully evade detection. *DroidKungFuUpdate* [29] is a notable example of dynamically executing payload going undetected with existing static analysis methods.

2.4.4 Analysis-aware Malware detection

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

Following the trends of the desktop platform, code obfuscation is evolving on Android [46, 47]. Obfuscation techniques are employed to:

- Protect the proprietary algorithm from rivals by making the reverse-engineering difficult.
- Protect Digital Rights Management (DRM) of multimedia resources to reduce piracy.
- Obfuscate the apps and make them compact, thus faster in execution.
- Hide the already known malware from anti-malware scanners so that it can propagate and infect more devices.
- Block or atleast delay human analyst and automatic analysis engines.

Dalvik bytecode is amenable to reverse engineering due to the availability of type-safe information such as class/method types, definitions, variables, registers, literal strings and instructions. Code transformation methods can be easily implemented on Dalvik bytecode with protection tool like *Proguard* [48]. *Proguard* is an optimization tool that removes unused classes, methods and fields. The meaningful class/method/fields/local variable names are replaced with unreadable information to harden the reverse engineering. *Dexguard* [49] is a commercial Android code protection tool. It can be used to implement code obfuscation techniques such as class encryption, method merging, string encryption, control flow mangling to protect an app from being reverse-engineered. Code transformation techniques can be used to hinder the malware detection approaches [46, 47]. Faruki et al. [50] proposed an automated Dalvik bytecode transformation framework to generate unseen variants of already known malware.

In the following, we cover various code transformation methods used to obfuscate the existing known malware. Some of these methods evade the existing disassembly tools [51]. Figure 2.5 illustrates a detailed outline of the prevalent code obfuscation and protection techniques:

2.4.4.1 Junk code insertion

Junk code or *no-operation code (NOP)* insertion changes the app size and evades the anti-malware signature database. The technique is popular due to its simplicity and ease of use. The Junk code insertion preserves the semantics of the original app by altering the opcode sequence. The opcode can be reordered with the *goto* instructions in-between the functions and alter the control flow, preserving the original execution semantics. These methods can be used to evade the signature-based or opcode-based detection solutions [46, 47]. Figure 2.6 illustrates junk code insertion technique to evade the analysis.

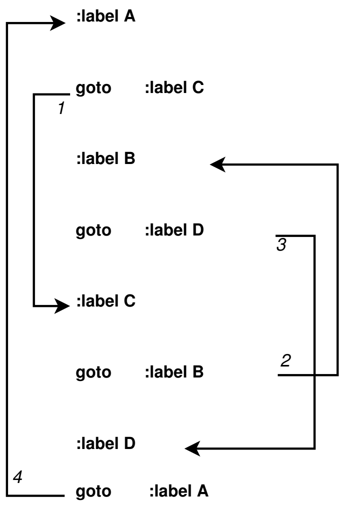


Figure 2.6: Junk code injection.

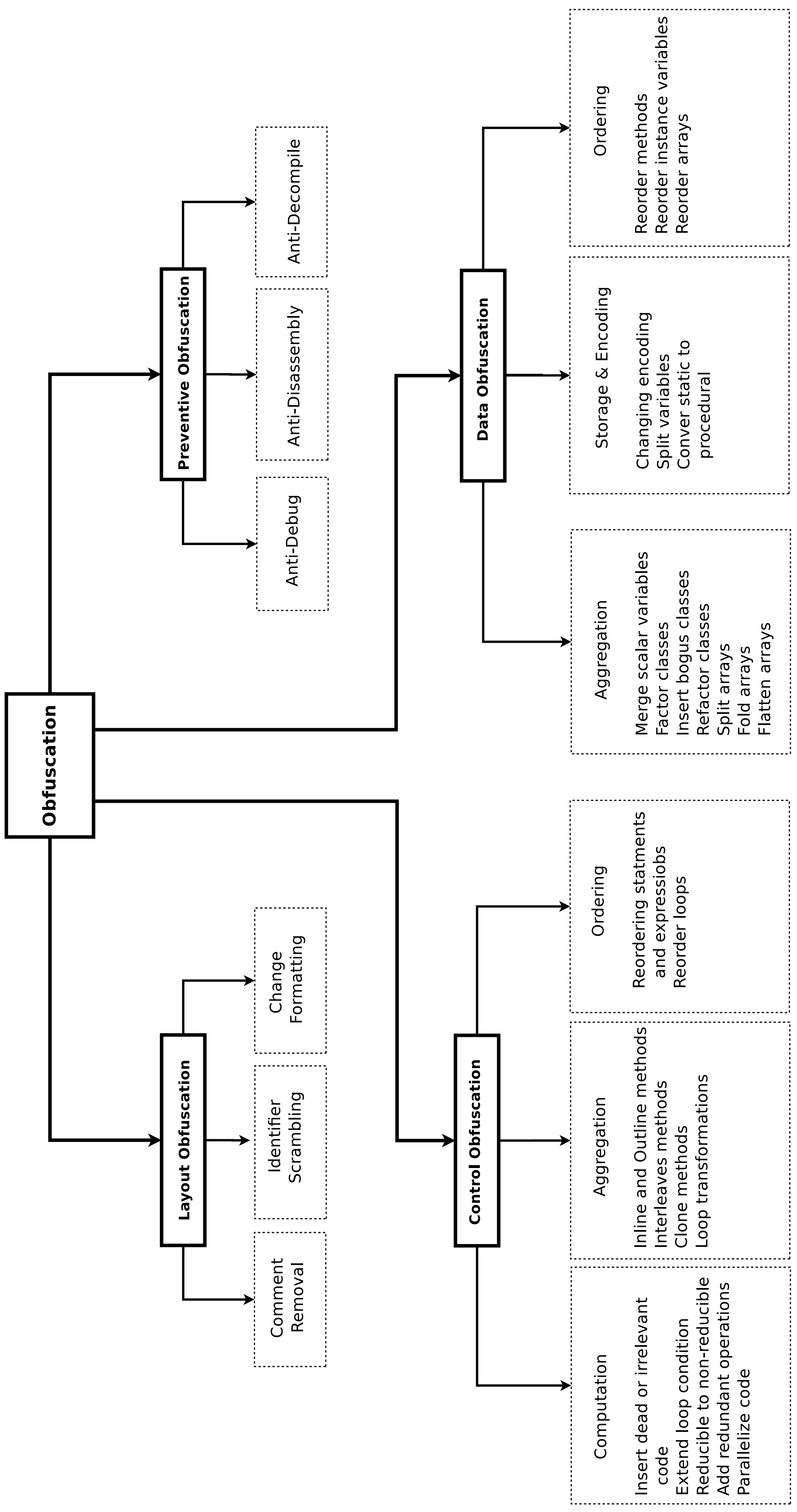


Figure 2.5: Obfuscation classification.

2.4.4.2 Package, Class, Method renaming

Android app has a unique package name. The Dalvik bytecode is typesafe, hence preserves the class and method names in the `dex` file. The commercial anti-malware use trivial signatures such as package, class or method names of known malware as its signature [52]. The trivial obfuscation like class or method renaming evades the existing signature-based detectors [47].

2.4.4.3 Altering Control-flow

Some anti-malware use semantic signatures such as control and data flow analysis to detect the malware variants employing simple transformation techniques [52]. The control flow of a program can be modified with (1) *goto* instructions; (2) insert and call the junk methods and (3) merging two or more methods into one. Though trivial, such techniques evade the commercial anti-malware [47]. Figure 2.7 illustrates the control-flow obfuscation.

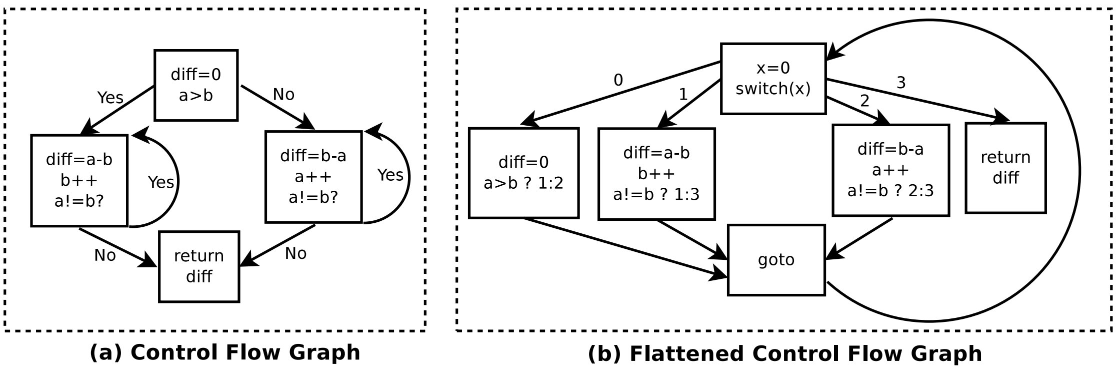


Figure 2.7: Control flow obfuscation.

2.4.4.4 String Encryption

The literal strings like messages, URLs, and shell commands reveal a lot about the app. The plain text strings can be encrypted and made unreadable to prevent the analysis. During the string encryption, multiple encryption methods (or keys) evade the decryption. The literal strings can be made available during the code execution, thus evading the static analysis. Listing 2.1 and 2.2 illustrate code snippets of original string and after the encryption.

```java
//original code

public void onClick(DialogInterface arg1,
 int arg2) {
 try {
  Class.forName("java.lang.System")
.getMethod("exit", Integer.TYPE)

.invoke(null, Integer.valueOf(0));
  return;

 } catch:(Throwable throwable) {
  throw throwable.getCause();
   }
}
```

Listing 2.1: Original snippet from [53].

```java
//String encryption

public void onClick(DialogInterface arg1,
int arg2) {
     try {
  Class.forName(C0n.  (GC0n.  [0xA],
C0n.  [0x09], GC0n.  [0xB]))
.getMethod(C0n.  (i1, i2, i2 | 6),
Integer.TYPE)
  .invoke(null, Integer.valueOf(0));
  return;
     } catch:(Throwable throwable) {
  throw throwable.getCause();
       }
}
```

Listing 2.2: after. snippet from [53].

2.4.4.5 Class Encryption

Decompilers tend to crash when the class names are too long or written in `non-ASCII` format [54]. Few malicious apps have demonstrated the same by using long class names to defeat reverse engineering tools. The technique is mentioned in [55], with new malware `Android/Mseg.A!tr.spy` reportedly employing class encryption. The relevant information such as product license-check, paid downloads, and DRM can be hidden by encrypting the entire classes [49].

2.4.4.6 Resource Encryption

Content of *resources* folder, *assets* and native libraries can be altered as unreadable, hence they must be decrypted at runtime [49]. For example, `Android/SmsZombie.A!tr` hides a malicious package in a JPEG file named `a33.jpg` in the assets directory [56]. Furthermore, `Android/Gamex.A!tr` conceals an encrypted malicious package in an asset named `logos.png` an image file [56].

2.4.4.7 Reflection API

Static analysis methods search sensitive Android API within the malware apps to map the malicious behavior. User apps permit Java reflection allowing the creation of class instances and method invocation with literal strings. To identify the class or method names data-flow analysis is necessary. However, the literal strings can be encrypted, making it difficult to identify the reflection API. Such techniques can easily evade static analysis approaches.

2.5 Malware Analysis and Detection

Android security solutions such as vulnerability assessment, malware analysis and detection techniques are divided into 1) Static; 2) Dynamic; and 3) Hybrid. Static analysis methods analyze code without executing. This may result in false positives. The dynamic analysis techniques monitor the executed code and inspect its interaction with the system. Though time-consuming they are effective for analyzing obfuscated malware. The Hybrid approach leverages the synergy of the static and dynamic analysis.

The security solutions can be categorized as rule-based [57] or feature extraction based machine-learning models [58]. Inappropriate feature selection can degrade the performance of the model and generate false-positives (i.e., false detection of benign as malicious). Moreover, the number of features under the problem must be small sized and effective as an On-device anti-malware. Feature reduction methods employing statistical measures such as *mean, standard deviation, chi-square, haar transforms* can be used to identify the prominent malicious attributes. The learning models can be developed by analyzing the processor, memory usage, battery consumption, system call invocation and network activities. Then, it can be used with the clustering or classification algorithms to predict anomalous behavior. In the following, we discuss some of the methods in detail.

2.5.1 Static analysis

Static analysis based approaches work by disassembly and decompilation without actually executing the APK. This approach is undermined by the use of various code transformation techniques discussed in Section 2.4.4.

2.5.1.1 Signature-based Malware detection

The existing commercial anti-malware employs signature-based malware detection by extracting interesting syntactic or semantic features [59]. The signature-based methods are time-efficient but, fail to detect unseen variants of known malware. Moreover, the signature extraction process being manual, its efficacy in the wake of the exponential malware outbreak leaves the device vulnerable for malware attacks. Another downside is the continuous increase in the size of signature database necessitates regular update. Furthermore, the exponential increased database size affects the time-efficiency.

2.5.1.2 Component-based Analysis

To perform detailed app security assessment or analysis, an app can be disassembled to extract the `AndroidManifest.xml`, resources, and Dalvik bytecode. The Manifest file stores important meta-data about such as a list of the components (i.e. activities, services, receivers) apart from the requested permissions, intent and intent-filters. The app security and assessment solutions can analyze the components using their definition and bytecode interaction to detect malicious app [60, 61, 62].

2.5.1.3 Permission-based Analysis

Requesting permission to access a sensitive resource is the central design of the Android security model. No application by default has any permission that can affect user security. However, identifying the requested dangerous permission alone is not sufficient to detect the malicious app [63, 64]. Sanz Borja et al. [65] employ <`uses-permission`> and <`uses-features`> tags in `AndroidManifest.xml` to identify malicious app. The authors utilized machine learning algorithms like Naive Bayes, Random Forest, J48 and Bayes-Net on a dataset of 249 malware and 357 benign apps. In [66] authors mapped the requested and used permissions from the manifest and their corresponding Dalvik bytecode API. The features are evaluated with machine learning algorithms on a reasonable size dataset. Enck et al. [67] developed *Kirin* to identify the combination of specific dangerous permissions to detect malicious apps.

2.5.1.4 Dalvik Bytecode Analysis

The Dalvik bytecode has rich semantic information. The classes, methods and instructions type information can be leveraged to identify malicious behavior. The control-flow and data-flow analysis can identify privacy leakage and telephony service misuse [57, 68, 69]. The control and data flow analysis are useful in rebuilding de-obfuscated bytecode to nullify the effect of trivial transformation [70].

Bytecode control-flow analysis identifies the possible paths that an app takes. The `jump`, `branch` and `method invocation` instructions alter the Dalvik bytecode execution. An intra-procedural (within a single method) or inter-procedural (i.e. spanning across multiple methods) analysis can generate a precise control-flow graph to identify the execution pattern. Karlsen et al. [70] formalized the Dalvik bytecode constructs and performed control-flow analysis to identify malware. For example, a malware app sending premium rate SMS to a pre-defined hard coded number can be detected with the constant propagation data-flow analysis [57]. Taint analysis, another type of data-flow analysis method tracks the uranine (colored) variables holding the sensitive information and track their flow within the program to identify privacy leakage [69]. The sensitive API-call tracing within the bytecode is useful in malicious behavior detection [71] and APK clones [72]. Zhou et al. [11] utilized the sequence of Dalvik opcodes to identify the repackaged malware.

2.5.1.5 Re-targeting Dalvik Bytecode to Java Bytecode

The availability of Java decompilers [73, 74, 75], Soot and WALA [76, 77, 78] based static analysis tools have motivated the researchers to re-target the Java source extracted from the Dalvik bytecode. Enck et al. [79] developed *ded* tool to convert Java source from the Dalvik bytecode. Later, they performed static analysis based control and data flow analysis of Java code with Fortify SCA framework [78]. In [80], authors developed *Dare* tool to convert the Dalvik bytecode to Java bytecode with 99% accuracy. Bartel et al. [81] developed the *Dexpler* plugin for *Soot* framework [76]. *Dexpler* converts the Dalvik bytecode to Jimple code. However, it is unable to handle the optimized dalvik executable (ODEX) files. Gibler et al. [82] employed *ded* and *dex2jar* [83] to convert the Dalvik bytecode into Java bytecode and source code respectively. WALA [77] framework identifies the privacy leakage within Android apps on a fairly large experimental data.

2.5.1.6 Privacy Leakage detection

Apposcopy [59], is a static analysis based semantic signature technique to detect apps stealing user sensitive private information. The proposed approach generates class and method level signatures to describe the malware family characteristics. Apposcopy signature matching algorithm combines static taint analysis and ICC based Call graph to detect inter-component-communication (ICC) properties. The authors evaluated proposed approach on a real-world Android apps to demonstrate the efficacy of Apposcopy. IccTA [84] leverages inter-component data-flow based static taint analysis to identify privacy leak among the app components. The author tracks uranine(tainted) variables called sources and trace them to the possible vulnerable functions called sinks to detect user-sensitive data ex-filtration. Proposed approach has high precision and recall values compared to the existing state-of-the-art. Furthermore, the proposed technique is useful in identifying inter-app based privacy leakage.

In [85], authors propose a reduced discovery of inter-component communication (ICC) on Android platform. Authors propose inter-component communication specifications based on location and substance. Experimental evaluation identifies more than 93% ICC locations with reasonable time. Furthermore, the Epicc tool identifies ICC vulnerabilities with low false positive compared to the existing techniques reported in literature. FlowDroid [86] is a *context-*, *object-* and *field* sensitive static analysis tool that formalizes the components life-cycle to detect privacy leaking apps. Flowdroid detects privacy invasion by explicitly tracking the data-flow analysis.

2.5.2 Dynamic analysis

Static analysis and detection techniques can be evaded by encrypted, polymorphic and code transformed malware. The dynamic analysis methods execute the app in a protected environment, providing all the emulated resources it needs, thereby learning from the interaction with system to identify malicious activities. Some dynamic analysis methods have been implemented, but the resource constraints of a smartphone limit such execution methods. Android app execution being event-based with asynchronous multiple entry points, it is important to trigger those events. User Interface (UI) gestures such as tap, pinch, swipe, keyboard, and back/menu keypress need to be automatically triggered to initiate the app interaction. Android SDK has inbuilt *monkey* [87] tool, to automate some of the gestures discussed above. To perform an in-depth monitoring, one may need to modify the framework by inserting the tracking code known as *Instrumentation*.

A serious drawback of the dynamic approach is that some malicious execution paths are not invoked on account of the missed events. If a malware triggers at a specified time, such event may not be captured and evades the analysis sandbox. Anti-emulation techniques such as Sandbox detection [88, 89], analysis environment timeout, delaying the malware execution evades the dynamic analysis. The dynamic analysis techniques can be classified as follows:

2.5.2.1 Profile-based Anomaly Detection

Malicious apps create Denial of Service (DoS) attacks by keeping hardware resources busy with unnecessary tasks. The parameters such as CPU usage, memory usage, network traffic pattern, battery usage may be an indication of such attack. The discussed parameters and system calls are collected from the Android subsystem. The analysis techniques use machine learning methods to distinguish the anomalous behavior [58, 90, 91].

2.5.2.2 Malicious Behavior Detection

Specific malicious behaviors like sensitive data leakage, sending SMS/emails, voice calls without user consent can be accurately detected by monitoring the particular features of interest [92, 93, 94, 95, 96].

2.5.2.3 Virtual Machine Introspection

Virtual machine introspection techniques regenerate the context of guest machine from the virtual machine monitor [97, 98]. This task is possible by extracting the data-structure information at the kernel. In [99], authors proposed a monitoring technique by mimicking the execution from outside the guest machine [99]. DroidScope [100] traverses the Android kernel data-structures to recreate the OS view. Furthermore, DroidScope generates Java and native app components simultaneously. The downside of app behavior monitoring from an emulator (VM) is, an emulator itself is susceptible against the malicious app that defeats the analysis purpose. Virtual Machine Introspection approaches can be employed to detect app behavior by observing the activities outside the emulator [100].

2.6 Deployment of Anti-malware

Security assessment, malware analysis and detection methods can be deployed at different places. Such assessment and analysis methods range from On-device solution to a completely Off-device or Cloud based analysis techniques.

2.6.1 On-Device

Signature-based malware is simple and efficient. The detailed assessment and analysis remains constrained on a mobile as compared to the desktop anti-malware. The lightweight risk assessment solutions may be devised by analyzing the components and permissions for an On-device solution [67]. Following are the limitations of an On-device anti-malware:

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

2.6.2 Distributed (On- and Off- Device)

On the fly analysis and detection is performed on the device; detailed and computationally expensive analysis can be carried out on the remote server. Profile-based anomaly detectors, measure resource usage parameters on the device; then send back to the server for detailed analysis. The analysis result is sent back to the device [58, 93]. The uninterrupted Internet availability and associated cost is a concern. In case of unavailability of network resources, host-based detection approach can protect the device from malware attack [91].

2.6.3 Off-Device

It is important to automate the deep static analysis of a new malware sample to enable the human analysts identify and mitigate the malware. Such automated deep analysis solutions need computational power and memory. Hence, they are usually deployed off-device [57, 69, 100, 28].

2.7 App assessment, Analysis and Detection

Industry and academia have proposed several solutions for Android malware analysis and detection. In this section, we survey and examine promising reverse-engineering tools and detection approaches.

2.7.1 Comparing analysis techniques

Table 1 compares different analysis and detection techniques to identify the limitations of existing state-of-the-art in “single malware” based on 3 parameters: (1) goal; (2) methodology; and (3) deployment. Androguard is a static analysis framework employed by many methods to detect malicious apps. Andromaly, Crowdroid and Paranoid Android use machine learning based anomaly detection to identify malicious apps based on benign properties. Table 1 compares the static and dynamic analysis techniques. The analysis of the existing state-of-the-art is detailed in Appendix A.

Tool	Goal			Methodology						Deployment			Availability
	Assessment	Analysis	Detection	Static	Dynamic	System call/API	Profile-based	Behavioral	VMI	On-Device	Distributed	Off-Device	
Androguard[52]	✔	✔	✔	✔								✔	Free
Andromaly[58]			✔		✔		✔				✔		Free
AndroSimilar[101]			✔	✔								✔	–
Andrubis[102]		✔	✔	✔	✔		✔	✔				✔	Free
APKInspector[103]		✔		✔								✔	Free
Aurasium[104]			✔		✔			✔				✔	Free*
CopperDroid[105]		✔	✔		✔	✔			✔			✔	Free*
Crowdroid[93]			✔		✔	✔		✔			✔		–
DroidBox[106]		✔			✔	✔		✔				✔	Free
DroidScope[100]		✔			✔	✔		✔	✔			✔	Free
Drozer[107]	✔			✔	✔						✔		Free/Paid
JEB [108]		✔		✔								✔	Paid
Kirin[67]	✔			✔						✔			Free
Paranoid Android[109]			✔		✔			✔				✔	–
TaintDroid[92]			✔		✔	✔		✔				✔	Free

2.7.2 Library based malware analysis

The third party app developers earn revenues on free apps by using the in-app advertisement libraries. The advertisement agencies provide the advertisement libraries to the app developers for inclusion to earn revenues, the only source of income for free apps. AdRisk [110] detected a few aggressive ad libraries performing targeted advertisements at the cost of user privacy. There have been instances of ad-affiliate networks getting classified as suspicious either due to: (i) targeted advertisements; (ii)sending malicious advertisements; compromising the user security [10]. Thus, it is equally important to detect such ad libraries within an app to make an informed decision. AdDetect [111] is a promising semantic approach that detects the presence of in-app ad library with reasonable accuracy compared to existing approaches.

2.7.3 Miscellaneous Techniques

Damopoulos et al. [91] proposed a combination of host and cloud-based Intrusion Detection System (IDS). In particular, authors highlight the importance of such a system to protect the smartphone. If the network resource availability is low, it performs host-based detection. If the device battery is drained, the prototype intelligently opts for the cloud-based detection to leverage the infinite processing and memory. In [112], authors proposed PERCEPT-V, a smartphone-based UI visual tags with a sampling algorithm for lighting, ambiance and usage angle environment.

SMS Trojans capable of sending messages to premium-rate numbers are growing to maximize monetary benefits. Elish et al. [94] devised a static anomaly detector to identify illegitimate data dependence between arguments of user input call-backs to sensitive functions. Using this approach they demonstrated the detection of some Android malware that send messages without user knowledge or consent. As the authors do not consider inter-component communication they fail to detect elusive SMS Trojans [113]. AsDroid [95] is a static analysis tool that detects stealth behavior leveraging semantic mismatch between the user-interface text and corresponding sensitive feature misuse.

2.7.4 Analysis Environment Detection

Vidas et al. [89] proposed a system to identify the analysis environment. The authors identify the behavior differences, performance evaluation, the absence of smartphone sensors and typical software capabilities. Such a system highlights the importance of employing analysis aware malware to reveal the hidden behavior. Faruki et al. [114] proposed a platform-neutral Sandbox to detect the stealth Android malware identifying the analysis system. Authors also propose a machine learning model to predict the resource hogger apps. Furthermore, [115] proposed a novel solution based on a behavior-triggering stochastic model to detect the targeted and advanced malware.

Rattazi et al. [116] proposed a systematic approach for identifying critical objects disallowing access control. Authors performed specific experiments to test their hypothesis and concluded that the newer capabilities are yet to mature for deployment on mobile internet devices. Petsas et al. [117] demonstrate advanced malware apps evade emulated environment hindering the analysis systems. Authors patched the existing malware apps with anti-analysis features and demonstrated the shortcoming of the existing frameworks [105, 118, 119, 120, 121, 122]. In [123], authors proposed a comparison framework for existing dynamic analysis Sandboxes and identified the limitations of automated malware analysis techniques. They concluded that advanced and targeted malware can evade existing sandbox approaches.

2.7.5 Comparing Web based analysis Tools

Table 2 lists existing Sandbox prototypes available as web services. Andrubis [102] and Copperdroid [105] are implemented on top of (i) Taintdroid [92], an on-device privacy leakage detector; and (ii) Droidbox [106], a dynamic analysis technique. The Mobile Sandbox [124] is an automated malware analysis and detection approach incorporating native code analysis, a facility unavailable with the existing prototypes. Droidanalyst [125] is an anti anti-analysis sandbox to detect analysis environment aware malware. Apps are classified as resource hoggers based on the data transmitted/received. The approaches using Taintdroid or Droidbox have to modify Android OS. The Appendix A elaborates popular static and dynamic tools and techniques available for Android malware analysis.

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

2.8 Summary

The Android mobile OS is a core delivery platform providing ubiquitous services. The monetary gains have prompted malware authors to employ various attack vectors to target Android. The massive increase in unique malware app signature(s) and limited capabilities within Android evades the analysis. The signature-based techniques are insufficient against unseen, cryptographic and transformed code. The researchers have proposed static, dynamic and hybrid analysis techniques to protect the centralized app markets. In this chapter, we discussed Android security architecture and its issues, malware penetration and various malware stealth techniques. In Section 2.5 we discussed relevant *static* and *dynamic* malware analysis techniques. Both approaches can be used separately with each incurring specific limitations.

The static analysis can be evaded by encryption, transformation and code protection techniques as discussed in Section 2.4.4. The dynamic analysis can be evaded by several anti-emulation techniques covered in Section 2.5.2. We also covered prominent malware analysis and detection approaches as summarized in Table 1 according to their goal, methodology, and deployment. We conclude that a single analysis technique can be evaded by a targeted malware. To identify a wide variety of new malware, a comprehensive evaluation framework incorporating the complementary static and dynamic analysis techniques can improve the analysis coverage. Based on the existing reviews in the chapter, we propose to integrate the synergy of multiple static and dynamic analysis techniques to improve the analysis coverage. We propose multiple techniques based synergic static and dynamic analysis framework to analyze the Android malware.

In the next chapter, we will discuss our proposal ApPRaISe, an Android permission analysis technique to detect over-privileged and malicious apps.

Chapter 3

ApPRaISe: Static Analysis of Android Malware

In the previous chapter, we discussed the Android security model and existing malware analysis techniques. The Android permission model secures the applications and protects sensitive devices resources. In this chapter, we propose ApPRaISe, an Android app Permissions risk classification technique. The ApPRaise statically extracts permissions declared in the Android manifest and important bytecode features to identify the risk that dangerous permissions pose if used inappropriately by the app developer. The proposed technique employs the declared permissions and their use within the Dalvik bytecode to determine over-privileged apps and susceptibility to their misuse. The goal of this chapter is to design an analysis technique capable of identifying the dangerous permission use and associate the risk posed in wake of ever increasing Android apps.

3.1 ApPRaISe: Overview

The Android permission model is an important security measure to prevent unauthorized access to the sensitive device resources. Permissions can be defined by the Android OS and third party app developers. The Android System defined permissions provides control over the sensitive information through the framework API to invoke protected resources (access device hardware, read device ID, modify device settings etc.). The Android Kitkat version 4.4 has defined 145 permissions, out of which 128 permissions are available to the app developers [1].

The Android permissions are defined with 4 protection levels associating a risk to necessitate permission usage at install time. Android security model classifies the permissions into (1) Normal; (2) Dangerous; (3) Signature; and (4) SignatureOrSystem. Among the four levels, 'Normal' group is granted by default; and 'Dangerous' permissions require explicit user approval at install time. The Android 'Dangerous' category perform sensitive functionality that costs money or acquires sensitive resources such as camera, phone, network or sending SMS. In respect of permissions, Android adopts either 'accept all' to permit app installation or 'accept none' to deny the installation. A naive user may not realize whether an app is requesting some unnecessary permission(s). The user may not be fully aware of the implications for the device security.

For an example, it is uncommon for a game app to request `SEND_SMS` permissions. A naive user, unaware of the consequences, may install the app requesting multiple dangerous permission approval. A single isolated dangerous permission alone may not be harmful. However, a combination of two or more dangerous permissions requested by an app need to be analyzed carefully. For example, an app either using `INTERNET` or `READ_SMS` permission may not pose a risk to the device. If the app requests both the permissions together, privacy leakage can happen as the SMS can be read and forwarded to other user(s) via Internet misusing the permissions pair.

Hence, we propose ApPRaISe, an app permission assessment model that identifies the risk associated with dangerous permissions. To improve the accuracy of risk assessment, presence of important bytecode features native, dynamic, cryptographic and reflection code observed more among the malicious apps are considered. We perform the $n-Set$ permission usage analysis and identify the risk associated with dangerous permissions. The proposed methodology consists of following steps:

1. Extract manifest binary and convert it to readable form.
2. Extract dangerous Permissions declared in the manifest.
3. $n - Set$ construction for benign and malicious dataset.
4. Frequency analysis to determine discriminatory $n - Sets$.
5. Identify contributing Permission $n - Set$.

6. Disassemble the Dalvik bytecode and map the declared Permission and its actual use in the Dalvik bytecode.

7. Identify over-privileged APK if inconsistent define-use mapping.

8. Identify risky and malicious apps based on permission usage.

In the following subsections, we shall be discussing the steps in detail.

3.1.1 n-Set Permissions Usage Analysis

In the first step, we extract the Androidmanifest binary from the APK and convert the manifest to readable form. Declared permissions are extracted from this file. This procedure is repeated for the benign and malicious dataset. For an example, app A manifest has requested `[INTERNET, SEND_SMS, READ_SMS]` permissions. $1-Set$ is constructed with `[INTERNET]`, `[SEND_SMS]` and `[READ_SMS]` permissions. $2-Set$ is illustrated by `[INTERNET SEND_SMS]`, `[INTERNET READ_SMS]` and `[SEND_SMS READ_SMS]` group. Figure 3.1 illustrates individual permissions requested by the benign and malicious apps respectively.

Similarly, $3-Set$ is constructed with `[INTERNET SEND_SMS READ_SMS]` permissions. Similarly, we generate upto 5-Set requested permissions from the manifest declared permissions. In the next step, we compare the frequency of analysis of benign and malicious apps for single permission ($1-Set$), combination of 2 permissions ($2-Set$) and combination of 3 permissions ($3-Set$) is evaluated against 8,341 benign Google Play [126] and 6,298 real malware from public repositories [2, 127, 128]. Figure 3.1 illustrates a comparison of different declared $1-Set$ permissions. $1-Set$ (Figure 3.1) lists `SEND_SMS`, `READ_SMS` and `READ_HISTORY_BOOKMARKS` requested by malicious apps in substantial number compared to benign apps.

Figure 3.2 illustrates 2-Set frequency distribution among benign and malware. The `[INTERNET READ_PHONE_STATE]` and `[INTERNET READ_SMS]` are requested more often by the malware as compared to benign apps. If the `[READ_PHONE_STATE]` is followed by `[INTERNET]`, the collected sensitive user information may be diverted to remote location. Similarly, the read SMS messages can be sent to the remote server. We identified top 70 permission combinations discriminating between the benign and malicious class. Furthermore, to improve the analysis accuracy, we

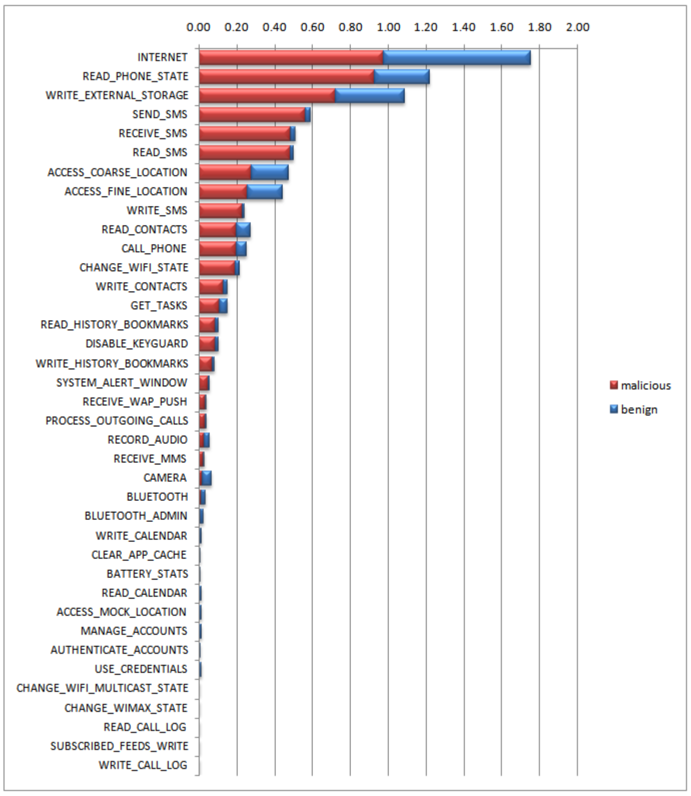


Figure 3.1: Comparative analysis for 1-Set permission.

have included the following Dalvik bytecode features, after experimentally evaluating 21,472 apps identifying their significant presence among malware.

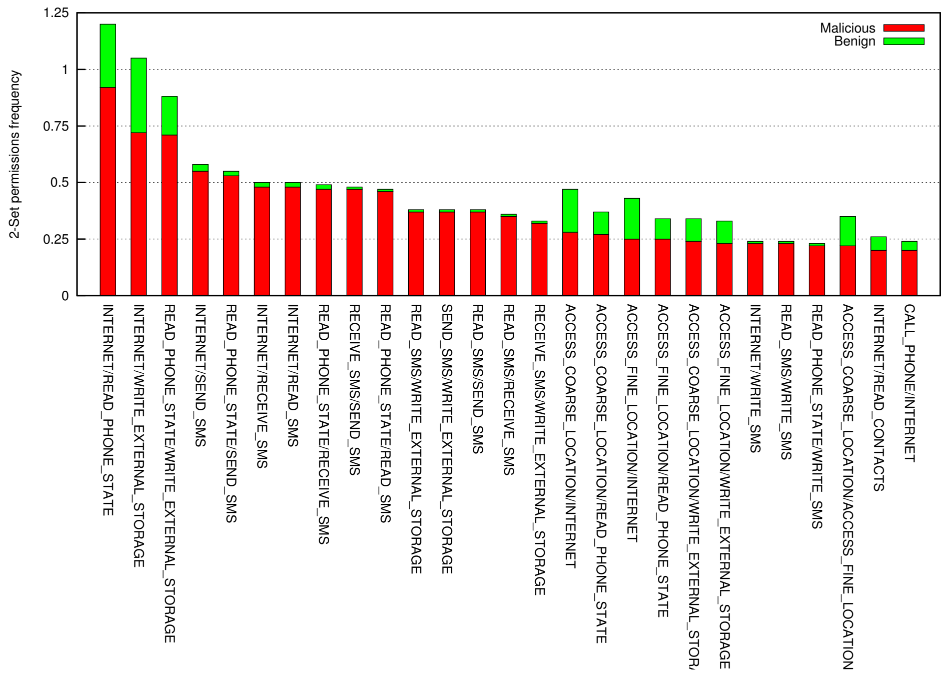


Figure 3.2: Comparative analysis for 2-Set permission.

1. **Cryptographic Code:** This feature is present if the strings are encrypted in an APK file.

2. **Native Code:** This feature is present if an app contains native (C/C++) code embedded as library or executable.

3. **Dynamic Code:** This feature is present if an app can load external Java classes at runtime.

4. **Reflection Code:** This feature is present if an app uses Java Reflection API in the code.

Figure 3.3 illustrates 3-Set frequency distribution among of benign and malicious apps. The permissions `[INTERNET READ_PHONE_STATE WRITE_EXTERNAL_STORAGE]` and `[INTERNET READ_PHONE_STATE SEND_SMS]` are the two top permissions requested by malicious apps compared to the benign. If the `[READ_PHONE_STATE]` is followed by `[WRITE_EXTERNAL_STORAGE]` and `[INTERNET]` permissions, the sensitive device information is first written to the external storage and ex-filtrated using `[INTERNET]` permission. Similarly, `[READ_SMS]` permission followed by `[WRITE_EXTERNAL_STORAGE]` and `[INTERNET]` can read the private user information and exfiltrate the data when Internet is available. The other permission set can be inferred from Figure 3.3.

3.1.2 Identifying Over-privileged APK

A developer declared dangerous permissions explicitly in the Androidmanifest. The security model ensures the declaration of dangerous permission in the app, if a sensitive resource need to be accessed by the APK. The mapping between the declared permission from the manifest and its actual use in the Dalvik bytecode identifies unused permissions. Based on the `androidmanifest.xml` defined and Dalvik bytecode use permission policy, an analyst can determine the over-privileged and malicious apps. In other words, an app must request a minimal set of permissions to perform the intended functionality. Each dangerous permission is associated with a corresponding Android API that restricts and controls the sensitive resource use [21]. To map the permission usage in the Dalvik bytecode, we need a relatively complete database that map permissions to their corresponding APIs. For example, we correspond a define and use only if `RECEIVE_BOOT_COMPLETED` permission

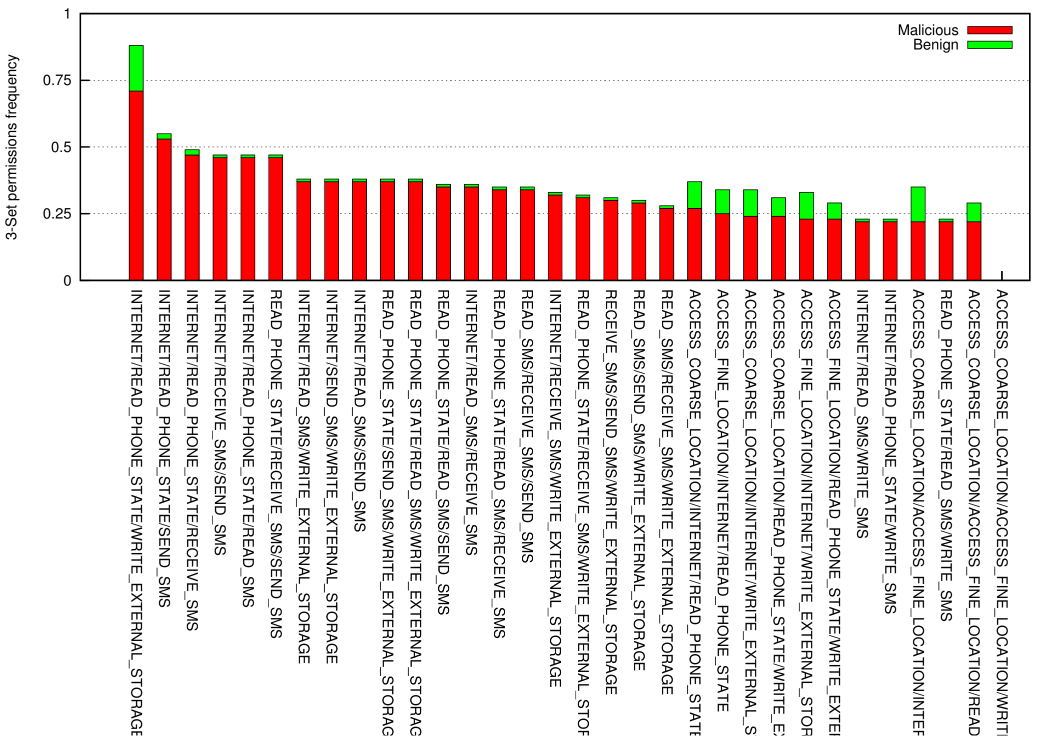


Figure 3.3: Comparative analysis for 3-Set permission.

receives the `android.intent.action.BOOT_COMPLETED` intent message. We leverage PScout [129] and manual mapping for permissions-to-API and Intent message mapping upto Android Kitkat OS version 4.4.

To identify the unused permissions ApPRaISe model performs the following steps:

1. Lookup API, intent-filters or content providers corresponding to the permission from the `androidmanifest.xml` file and map the use of Dalvik bytecode in `classes.dex`. If no reference of declared permissions is found in the Dalvik bytecode, we conclude that the permission is unused, else run step 2. The presence of unused permission imply the app under inspection is over-privileged.

2. Perform reverse path reachability using synchronous and asynchronous control flow and locate the bytecode where the permission is used. For each method invoked in the Dalvik bytecode, we compare the API name with the method.

3. If the topmost method found in the path is an entry point method called by the Android framework (e.g., onClick, onCreate, onBind), we conclude

that permission declared in the manifest file has been used in the app code. Otherwise, permission is treated as declared but unused. In the latter case, the APK is labeled as over-privileged.

3.2 Experimental Evaluation

The manifest permission features for benign and malicious apps are extracted and evaluated against a classification model by collection and experimental evaluation of nearly 20,000 apps collected from Google play and real malware from known repositories. In the first experiment, we extracted manifest permissions from the benign and malicious dataset. Then, we ranked the permissions and their $n-Set$ based on their usage in malicious compared to benign. In the second experiment, we analyze and map the use of declared permission in the manifest file.

3.2.1 Dataset Preparation

Table illustrates the dataset to classify the $n-Set$ permission model. We crawled 32 categories of Google Play and selected top apps from each benign category. The malware apps are gathered from sources listed in the Table below:

Source	**# of Apps**
Malicious Apps	**6298**
Malware Genome Project [2]	1260
Contigiominidump [127]	654
VirusShare [128]	3784
Benign Apps	**8341**
Google Play [126]	7235
Asian Third Party markets [130, 131]	1106

Table 3.1: Dataset for ApPRaISe model.

3.2.2 Performance Evaluation

We evaluated the ApPRaISe model on 73 features (permissions and bytecode features) extracted from 11,639 apps and trained them with the RandomForest [132] decision tree classifier. RandomForest is selected to balance the performance trade-off. Permission modeling results are illustrated in Table 3.2, enlisting acceptable low false positives.

To improve the efficiency of ApPRaISe, we reduced feature space through a feature selection method. We employed minimum redundancy Maximum Relevance (mrMR) [133]. The mrMR removes the redundant features and incorporates the relevant features to improve analysis efficacy. We selected top 20 features to investigate the effect of reduced feature space on the accuracy of the approach. Table 3.3 illustrates that 85% accuracy can be achieved by identifying risky apps based on permissions and important bytecode features with an acceptable false positive rate.

Actual Class	**Predicted Class**	
	Malware	**Benign**
Malware	**81.3%**	18.7%
Benign	2.8%	**97.2%**

Table 3.2: Performance of ApPRaISe on 73 features.

During app evaluation, we observe that `ACCESS_NETWORK_STATE` permission is paired with the `INTERNET` permission. More than 72% apps requested `[ACCESS_COARSE_LOCATION ACCESS_FINE_LOCATION]` permission set. Even the benign wallpaper APK requested both permission necessitating justification. Some applications required excess permission such as reading the browsing history, bookmarks, read phone storage, even though the app category did not necessitate such action. Hence, the ApPRaISe developed $n - Set$ analysis technique to identify risky and malicious apps.

Actual Class	**Predicted Class**	
	Malware	**Benign**
Malware	**84.6%**	15.4%
Benign	7.7%	**92.3%**

Table 3.3: Experimental evaluation: ApPRaISe with top 20 features.

3.3 Inference and Discussions

The proposed ApPRaISe evaluation answers the following:

- Is the distribution of permissions set among benign and malicious apps significantly different ?
- Is the proposed technique capable of filtering benign apps from further scrutiny ?
- Can permission be the sole criteria to classify malicious apps from benign
- Reasons for false alarms.

In this section, we analyze the proposed app classification model and reasons for inaccuracies in the analysis.

3.3.1 Security Analysis

The ApPRaISe evaluation shows malicious apps use `INTERNET` `ACCESS_WIFI_STATE` more frequently compared to benign, the permission pair is useful for sending sensitive information over the network proxy. The `READ_PHONE_STATE` is also used more often, as one can access the device-id, Phone number and `SIM ID`. Furthermore, `READ_LOGS`, `ACCESS_FINE_LOCATION` and `ACCESS_COARSE_LOCATION` are more frequently visible among malicious apps. We also identify `INTERNET` following `READ_PHONE_STATE` permission used more often by malicious apps to send sensitive device information to a remote host.

The `INTERNET` is the most used permission among the free Google Play apps. The main reason is the inclusion of third party libraries for advertisement revenues. Thus based on a single permission the analysis technique incur False alarms. If one tries to reduce False Positive (FP), the malware will be missed by the detector. Hence, our proposed technique consider the $n - Set$ permission analysis and maps the manifest declared permissions against the Dalvik bytecode to reduce the false alarms. We have identified top 20 permission pairs requests occurring more often among malicious apps illustrated in Figure 3.2

The proposed ApPRaiSe is useful in detection of over-privileged apps and malware based on the declared and used $n - Set$ permission analysis. The mapping of defined permissions with their actual use within the Dalvik bytecode, identifies the surplus permissions. Based on the surplus permissions and its protection level, we identify whether the app can be potentially misused by other malicious applications. We infer the define and use permission to analyze the potential misuse of existing benign applications. The proposed model identifies over-privileged benign apps using permissions not actually required for the normal functioning. However, the proposed approach incurs false positive even if permissions and important bytecode instructions are used for analysis. The proposed model captures the def-use permission relationship of an APK under inspection. A possible source of inaccuracy arises when the inspected APK declares a superfluous permission and makes use of the particular permission.

For example, we observe `ACCESS_NETWORK_STATE` permission is paired with the `INTERNET` permission. More than 72% apps requested `ACCESS_COARSE_LOCATION` and `ACCESS_FINE_LOCATION`. Some applications required excess permission such as reading the browsing history, bookmarks, read phone storage, even though the app category did not necessitate such action. Furthermore, we have experimentally evaluated larger permission set of 28,946 malicious apps and identified the permission pairs used more frequently among malware. The legitimate apps may also require some pairs of dangerous permission raises the possibility of false alarms. For example, legitimate calendar app has declared `INTERNET` permission not used in the code. The component using the app has not defined it. Hence, a malicious app can misuse the vulnerable app.

3.3.2 Google Play app Category as analysis Feature

Google Play, the official Android app market allows the users to browse free and paid apps distributed into 32 categories (Books, Games, News etc.) We performed experimental evaluation to identify the prevalent use of particular permissions within a specific category. For example, Figure 3.4 illustrates top permissions requested by *Tools* category apps. The figure illustrates `INTERNET`, `ACCESS_NETWORK_STATE` and `WRITE_EXTERNAL_STORAGE` at the top.

In case of the free Google Play apps, Dalvik bytecode analysis reveals the permission requests are on account of third-party advertisement networks and an-

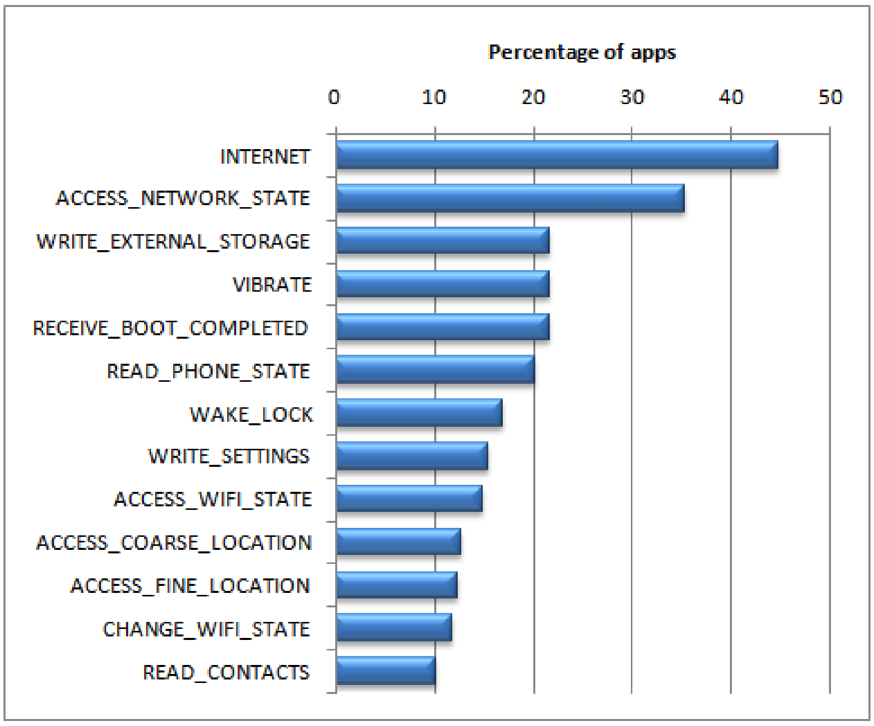


Figure 3.4: Requested permissions for Google Play *Tools* category.

alytics services. Hence, we define a parameter θ to depicts the minimum percentage apps requesting a particular permission within same category. Figure 3.4 illustrates $\theta = 20$ set for *Tools* category requesting `VIBRATE` permission. The ApPRaISe model identifies the app as non-risky. If the same app additionally requests `ACCESS_FINE_LOCATION` permissions, then ApPRaISe identifies it risky as only 13% (less than 20%) apps request that permissions within *Tools* category. A *permission that is requested by more than θ percent of apps within some category is permissible for the app in same category.* However, the *Tools* category apps do not use the above permission.

Figure 3.5 illustrates permissions requested by *Communication* category apps. The *Communication* category frequently requests `SEND_SMS` and `READ_SMS` *permission* necessary for communication functionality. We have experimentally evaluated benign apps with several θ values as illustrated in Table 3.4. To decide suitable θ, we analyzed permissions usage for each benign category. The *communication* category `READ_SMS` permission is requested by 16% apps and `SEND_SMS` by 22% apps. The SMS read/write functionality falls under under *communication*, which suggests that $\theta = 15$ is suitable for this category.

We set ApPRaISe experimental $\theta = 15$ to balance the apps risk and FP. The

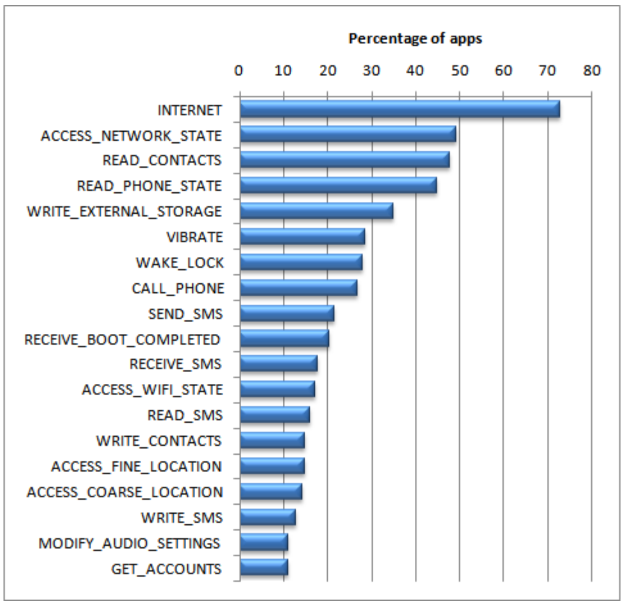


Figure 3.5: Percentage permissions: *Communication* category.

inclusion of app category features reduced to **5.65%** from **7%**. Table 3.3 illustrates 7.7% FP. To lower the FP, ApPRaISe analyzed benign app category as a feature.

θ **Value**	**–NA–**	**25**	**20**	**15**	**10**
FP Rate	7.00%	6.64%	6.03%	5.65%	5.25%

Table 3.4: ApPRaISe FP rate for θ values.

3.3.3 Inaccuracy in Permission Classification

Over-estimating the permission fails to identify potential malware. Some permission pairs may be necessary for legitimate app functionality. The similar pairs being used by malware generates False Negative (FN). For example, an app using `INTERNET` or `READ_SMS` permission individually may not pose a risk to the user device. However, if the app requests both the permissions together, the permissions pair can be misused. Conversely, under-estimating the triggers may cause false positives. For instance, legitimate APK uses a permission pair more frequently observed in malicious apps. Hence, the genuine app is identified malicious.

3.3.4 Comparing Permission based App Analysis

The Permission based access control to access sensitive resources (SMS, Call, Camera, WiFi etc.) is core of Android security model. However, naive users cannot judge the appropriateness of permissions and do not have the privilege to reject a single dangerous permission [134, 135]. Mapping the manifest permission requests, it may not possible to identify a malicious app. Nevertheless, permission is an important feature to identify the risk associated by granting them all [63, 64].

Sanz Borja et al. [65] used <`uses-permission`> and <`uses-features`> tags present in `AndroidManifest.xml` file as features. They applied machine learning algorithms Naive Bayes, Random Forest, J48 and Bayes-Net on a dataset consisting of 249 malicious and 357 benign apps. To demonstrate that such permission based analysis model is not always robust, we experimentally evaluated the modified malware dataset (by adding benign app permissions to malware) with the classification algorithm again. In each malicious app manifest, we declared top 5 permissions of the benign dataset and the remaining content is kept unmodified. When the features are evaluated with the WEKA classifiers, the False Negative Rate (FNR) (FNR: malware missed by detector) increases and classification rate drops to 71%.

Huang et al. [66] evaluated the requested permissions and the manifest meta-data features using machine learning algorithms on a dataset of 1,25,249 malicious and benign apps. Enck et al. [67] developed permission certification tool Kirin consisting a set of rules regarding combination of permissions in their database. The app installation is rejected, if some app requests a certain combination of permissions considered dangerous in the rule database. However, the rules need to be manually added into the Kirin according to the requirements. Kirin also provides nine sample security rules to mitigate malware. The proposed ApPRaISe is an automated $n - Set$ based permission classification technique to associate permission pair risk based on dangerous permission use. For example, `READ_PHONE_STATE` can access the `Device-ID`, Phone number and `SIM ID`. Furthermore, `INTERNET` followed by `READ_PHONE_STATE` is handy to send sensitive information over the network.

Sarma et al. [63] proposed several risk signals based on permissions requested by apps that are useful in alarming users. In particular, Category-based Rare Critical Permissions signal, denoted CRPC(θ), is triggered when any app requires a critical permission that is requested by less than θ percent of apps in the same category.

In the same way, Rare Critical Permissions signal, denoted RPC(θ), is triggered when an app requests a critical permission that is requested by less than θ percent of all apps. However, dataset of just 121 malware is very primitive in order to understand their permission request patterns in comparison to benign apps.

Felt et al. [21] observed that developers often declare extra permissions. They developed *Stowaway* that identifies extra permissions and helps developer rectifying their mistakes. They evaluated 940 Android apps and reported 33% over-privileged apps. Au et al. [129] built a version independent *PScout* tool that extracts permission specification from the Android OS source code using static analysis. *PScout* generates a more complete permission specification than *Stowaway*. We build on the existing techniques to further improve the analysis

We observed previously that permissions in Android are coarse-grained in nature. Jeon et al. [136] proposed a new approach to understand and implement fine-grained permissions on the Android OS. They presented *RefineDroid*, *Mr. Hide*, and *Dr. Android*, tools that infer and implement finer-grained permissions on Android without requiring platform modifications.

3.4 Summary

In this chapter, we proposed ApPRaISe to automatically analyze the Dangerous permissions declared and used among Android apps. The proposed technique achieves thorough analysis of permission distribution and its potential misuse. Unlike the existing approaches, we perform fine-grained analysis to map the defined manifest permissions to their actual use in Dalvik bytecode. We have experimentally evaluated Android permission define-use mapping to identify over-privileged and malicious apps.

The proposed model can analyze large number of apps to automatically detect over-privileged and malicious apps. However, repackaged malware may evade the analysis as the transformation does not remain limited to the `androidmanifest.xml`. The permission based analysis techniques can be evaded with Dalvik bytecode modification. To overcome the limitations and detect repackaged malware, we propose a byte-level file similarity approach AndroSimilar in the next chapter.

Chapter 4

AndroSimilar: Robust Statistical Signature for Android Malware

In this chapter, we propose AndroSimilar, a robust statistical feature signature to detect repackaged malware and unseen variants of known malware families. We characterize a bloom filter based file signature to detect repackaged and unseen malware variants. Furthermore, we experimentally demonstrate the efficacy of AndroSimilar in identifying obfuscated malware.

4.1 Threats and consequence of Repackaging

The current Android anti-malware use Signature-based detection techniques. The commercial anti-malware has limited capabilities due to Sandboxing on Android. The anti-malware solutions does not enjoy platform level privileges similar to Desktop OS. This necessitates alternative techniques to counter the zero-day malware. The ever increasing malware instances render the manual analysis impractical and infeasible.

The malware employs code obfuscation and app repackaging techniques and distribute malware towards less monitored third-party app markets [10]. Repackaging is a process of inserting malicious payload inside a benign app using reverse engineering techniques. The conventional signature-based approach has worked effectively to prevent known malware families. However, they can be evaded by trivial code transformation techniques.

4.2 AndroSimilar: Overview

In this chapter, we present a robust statistical approach that explores byte features [137] for capturing code homogeneity among variants of known malware families. We assume that variants of an existing family share some common attributes. The obfuscated malware variants are marginally dissimilar from each other. The minor dissimilarity is enough to evade the existing signature-based anti-malware. The proposed approach captures code similarity by identifying the rare features only present in the variants of families or related files. AndroSimilar, a byte-level approach generates variable length signatures to detect unseen, zero-day samples crafted out of the known malware.

4.3 Proposed Methodology

Figure 4.1 illustrates the AndroSimilar signature generation approach based on the hypothesis that two unrelated files have a low probability of having common features. The fixed-size byte sequence features are extracted based on the empirical probability of occurrence of their entropy values. The values are computed in a sliding window fashion. The popular features are identified according to their neighborhood rarity [137]. When two unrelated files share some characteristics, the features are considered weak generating false positives [138]. Initially, we generate signatures of known malware families in the existing representative malware database. Then, we compare the similarity score of an unknown app with the existing malware signature database. If the signature from database match with the unknown app beyond an experimental threshold, the application is labeled malicious. The proposed methodology is discussed below:

1. Submit Google Play, third-party, and obfuscated malicious app as input to AndroSimilar.
2. Generate entropy values for every byte-sequence and normalize them in $[0, 1000]$ range.
3. Select statistically robust features according to the similarity digest scheme as a representative to the app.

4. Store extracted features into Bloom Filters. The sequence of Bloom Filters is a signature of an app.

5. Compare the signature with the database to detect the match with known malware family. If the similarity score is beyond a given threshold, mark it as malicious (or repackaged) sample.

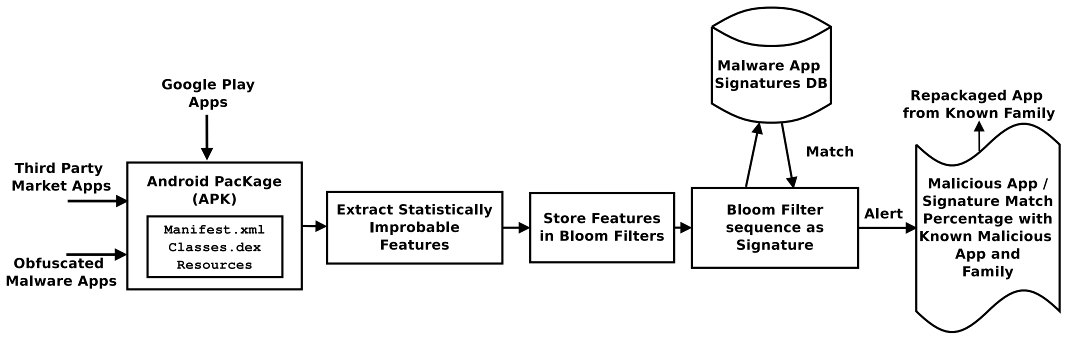

Google Play Apps
Third Party Market Apps
Obfuscated Malware Apps
Android PacKage (APK)
Manifest.xml
Classes.dex
Resources
Extract Statistically Improbable Features
Store Features in Bloom Filters
Bloom Filter sequence as Signature
Malware App Signatures DB
Match
Alert
Repackaged App from Known Family
Malicious App / Signature Match Percentage with Known Malicious App and Family

Figure 4.1: Proposed approach.

4.3.1 Normalized Entropy

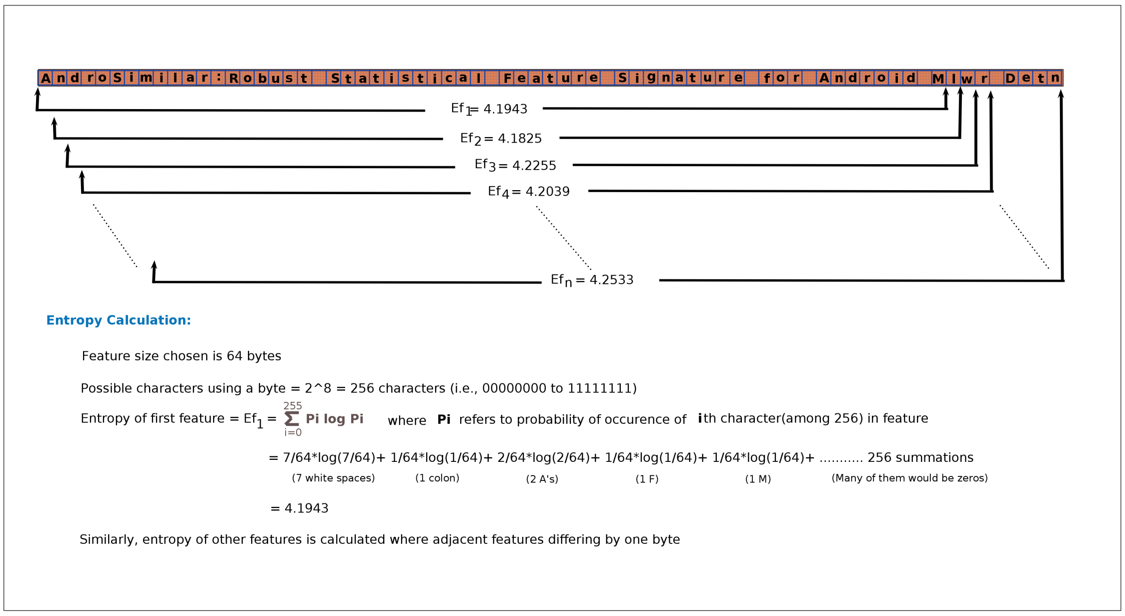
Entropy is a useful tool for extracting robust statistical features [138]. As single byte (8-bits) can have a `ASCII` in range of [0...255], we approximate the probability P(n) as occurrence of n in a given sequence of bytes B. For example, in a byte sequence $[23, 45, 23, 78, 54, 23, 90, 23]$, $P(23) = 4/8, P(45) = 1/8$ and $P(30) = 0/8$. Once the probabilities have been computed for all the values in range [0...255], entropy of the byte sequence is given as:

$$H(S) = \sum_{i=0}^{255} P(X_i) \cdot \left(\frac{1}{\log(P(X_i))} \right)$$

values of $X_k \in [0...255]$ and $X_i \neq X_j \forall i \neq j$. The proposed approach normalizes the entropy into a range 0 to 1000 using:

$$H_{normal} = \left\lfloor 1000 \cdot \frac{H(S)}{\log_2 B} \right\rfloor$$

B is 64 bytes, the size feature block for robust feature selection. Figure 4.2 illustrates the entropy calculation for $B = 64$. Successive entropy values are computed



Entropy Calculation:

Feature size chosen is 64 bytes

Possible characters using a byte = 2^8 = 256 characters (i.e., 00000000 to 11111111)

Entropy of first feature = $Ef_1 = \sum_{i=0}^{255}$ **Pi log Pi** where **Pi** refers to probability of occurence of **i**th character(among 256) in feature

= $\underbrace{7/64*\log(7/64)}_{\text{(7 white spaces)}}+ \underbrace{1/64*\log(1/64)}_{\text{(1 colon)}}+ \underbrace{2/64*\log(2/64)}_{\text{(2 A's)}}+ \underbrace{1/64*\log(1/64)}_{\text{(1 F)}}+ \underbrace{1/64*\log(1/64)}_{\text{(1 M)}}+ \underbrace{\ldots\ldots\ldots \text{ 256 summations}}_{\text{(Many of them would be zeros)}}$

= 4.1943

Similarly, entropy of other features is calculated where adjacent features differing by one byte

Figure 4.2: An example: Entropy calculation.

for sequences shifted by one byte. The entropy difference of one byte change due to related content has negligible variation. Normalizing the entropy values into a much larger range, $[0 \cdots 1000]$, enhances the difference and facilitates robust feature extraction. 0 is the minimum entropy and 1000 is the maximum entropy value. The entropy distribution between $[0 \cdots 1000]$ is the probability of occurrences of each normalized entropy value (i.e., occurrences/total). When entropy is measured for byte sequences of smaller size, variation is reduced. Hence, the features are not easily distinguishable and incurs longer processing time [138]. If a big length byte sequences is considered less features are obtained.

As illustrated in Figure 4.2, normalized entropy stream is generated for given byte sequence spanning an app in sliding window fashion. In our case, the window size is $B = 64$ bytes, and slide occurs by 1 byte. The Sliding window allows us to calculate the normalized entropy of a byte sequence using the value of previous byte sequence. Some apps may also consist of repeated sequence of similar byte-code. Such attribute occurrence results in False Positives (FP). The removal of ambiguous features helps to reduce the false positive without affecting the feature selection. The entropy-based statistical analysis approach suffers from false positive. The proposed methodology considers the rare statistical features that represent the similarity between two related files to reduce the FP.

4.3.2 Precedence Rank Values

To find the rare features from normalized byte sequences, a Precedence Rank R_{prec} is assigned to each normalized entropy value, such that former is proportional to latter's probability of occurrence [138]. The precedence rank generation maps a lower probability value to a higher rank (1 to 1000) and higher probability value to a low rank. The precedence rank calculation is like calculating a reverse exponential function. We apply, $1000 * (1 - exp(-prob * 1000))$ function for the same. To generate R_{prec} table, we need to analyze entropy distribution of benign and malicious APK files. Figures 4.3 and 4.4 illustrate normalized entropy distribution with B=64 and 128 bytes for a random set of benign and malicious apps. The benign entropy distribution is illustrated with blue and malware distribution is red color in the graph.

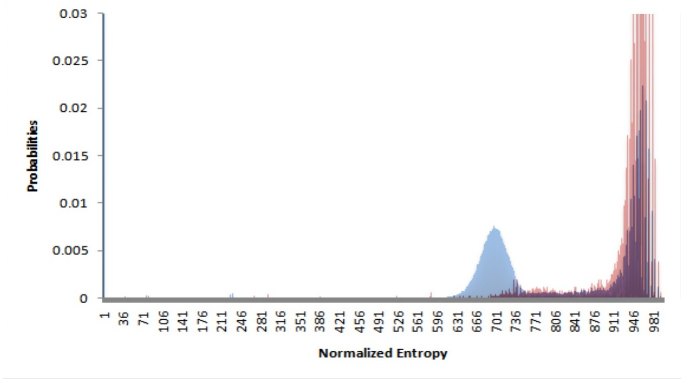


Figure 4.3: Normalized APK entropy distribution for B=64.

We can see high-peak at lower entropy values indicating the occurrence of repeated byte sequences, essentially weak features. The values are computed on the relevance of a given entropy value to differentiate malicious app from benign. Similarly, peak at the higher entropy values depicts compression tables with high randomness, indicative of weak features. The weak features do not discriminate between benign and malware; contributing towards false alarms. Removing such weak features reduces false-positives and improved feature selection. We assign R_{prec} to each normalized entropy in the range of 1 to 1000 by eliminating weak features. The lower R_{prec} indicates least likely discriminant feature. An example of R_{prec} stream is illustrated in Figure 4.5.

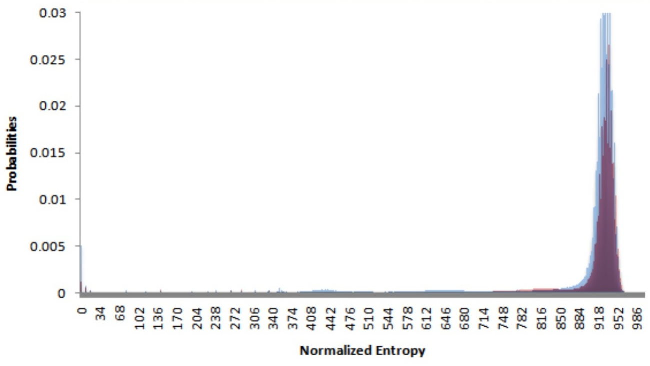
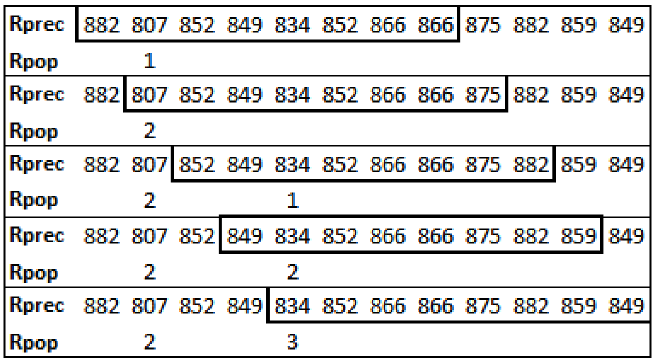


Figure 4.4: Normalized APK entropy distribution for B=128.

4.3.3 Statistically Robust Features

Considering a complete R_{prec} may not successfully extract rare features only present in two related files. Hence, we extract persistent local minima to pick the robust features. A local minimum is determined in a sliding window fashion. As stated earlier, lower the value of R_{prec}, higher the ranking in feature discrimination. To avoid superfluous features, we consider only those minima that persist among the multiple adjacent windows. After generating R_{prec} stream, it is not just important to consider features with lowest ranks. For each R_{prec} value, we maintain a popularity score R_{pop}. We consider a window size of W slid over the R_{prec} stream. In each window, left-most minimum R_{prec} is selected and its R_{pop} is incremented. Thus, if R_{pop} of a feature is k, $(k \in [1 \cdots W])$, it is the local minimum within a window of size $W + k - 1$, which further emphasizes the relative rarity of that particular feature [139]. For illustration, let us consider Figure 4.5.



Rprec	882	807	852	849	834	852	866	866	875	882	859	849
Rpop		1										
Rprec	882	807	852	849	834	852	866	866	875	882	859	849
Rpop		2										
Rprec	882	807	852	849	834	852	866	866	875	882	859	849
Rpop		2			1							
Rprec	882	807	852	849	834	852	866	866	875	882	859	849
Rpop		2			2							
Rprec	882	807	852	849	834	852	866	866	875	882	859	849
Rpop		2			3							

Figure 4.5: Robust statistical features.

As illustrated, R_{prec} stream of corresponding features of length B-bytes is shown in each row. Window W spans consecutive R_{prec} values and left-most lowest one is selected. In the first row, this minimum corresponds to $R_{prec} = 807$. The R_{pop} of the feature with $R_{prec} = 807$ is incremented in window shifted to the right. Next window (in Row 2) picks up the local minimum with $R_{prec} = 807$ and increments the corresponding R_{pop} by one again. The procedure continues until the R_{pop} values of all the local minima are calculated. The mapping of a "local minimum to a feature occurs only if the R_{pop} is above a particular threshold value". The threshold for R_{pop} is considered such that it enables collection the of statistically robust features [139]. For the above example, if the minimum popularity threshold is 3, we consider the feature with $R_{prec} = 834$ as statistically robust. Algorithm 1 illustrates AndroSimilar signature generation.

Algorithm 1: AndroSimilar signature generation.

```
Input  : mal_sign_db – Signature database.
         appset – Apps to be checked.
         threshold – Similarity threshold.
Output: malicious_set – Set of tuples as (appName, familyName).
1  foreach app in the appset do
2      score_results ← an empty array;
3      app_sign ← generate_improbable_features_sign(app);
4      foreach malicious_sign in the mal_sign_db do
5          score ← compare_signatures(app_sign, malicious_sign);
6          score_results.add(score);
7      end
8      best_similarity_score ← find_best_score_match(score_results);
9      if best_similarity_score >= threshold then
10         tuple ← (app.name, best_similarity_score.familyName);
11         malicious_set.add(tuple);
12     end
13 end
```

The robust features selected using above technique are hashed and then inserted inside bloom filters for compression. To aid efficient searching, 256 features are inserted in a single bloom-filter. The sequence of bloom filters becomes a statistical signature of the app. Two signatures of similar apps are then compared to find a similarity score ranging [0..100], with 0 being not similar and 100 as complete match [139]. The features stored within the Bloom filters are put together to generate the AndroSimilar signature. The robust feature signature generated from the APK file is variable length on account of unique features extracted from an app.

4.4 Signature-Set reduction

SDHash [138] signature is 2-3% of the actual APK file. The increased number of Android malware necessitate a signature for each malware. Instead of maintaining one signature for each malware, the proposed approach is capable of extracting a family signatures from multiple variants of a known malware family. This is done by clustering the same category app (malicious/benign) on the basis of similarity. The representative or a family signature is selected based on app signatures having more statistically robust features than any other app signature in the cluster, discarding the signatures of other family members.

The clustering within each malware family is performed with SDHash similarity. The distances between sample signatures in a single cluster is small (i.e., high similarity). The minimum value is chosen based on an empirically chosen *inter-app similarity threshold*. From each cluster, we select a single point capable of representing all the apps in the cluster. The dissimilar apps are represented as a separate cluster.

For an example, let us consider eight apps A_1, A_2, ..., A_8 that are variants of malicious family A with similarity score listed in Table 4.1. The underlying assumption is, the similarity score of app signature for the variant of same family is higher than the those from a different family.

SDHash Similarity Score(%)	**App Name**	**App Name**
70	A_1	A_2
65	A_1	A_3
90	A_2	A_3
94	A_4	A_5
97	A_5	A_6
20	A_2	A_5
30	A_1	A_6

Table 4.1: Similarity between app family.

4.4.1 An Example

To simplify the analysis, we map these feature similarity as a distance vector. Table 4.1 illustrates the signature similarity calculated by AndroSimilar signature algorithm based on SDHash similarity. The AndroSimilar signature $(App_1, App_2) = 70$, $(App_1, App_3) = 65$, $(App_4, App_5) = 94$ and so on. The signature similarity suggests that 70% (App_1, App_2) features are similar. To calculate the family signature, we find the distance between the apps within the cluster. The distance parameter is inversely proportional to similarity. We calculate the dissimilarity among the apps to remove least similar apps, retaining the most similar features among the closest apps. We calculate the distance score as:

$$Distance(App_1, App_2) = 100 - SimScore(App_1, App_2)$$

Table 4.2 values calculated from the AndroSimilar signature similarity illustrates the distance matrix of the apps from malware family A. As given in the Table, app A_1 is 100% similar to itself. Hence, the similarity score $SimScore(App_1, App_2)$ is 100. The distance formula, $Distance(App_1, App_1) = 100 - SimScore(App_1, App_1)$ gives 0. The first row in Table 4.2 maps (App_1, App_1) result as 0. Similarly, further mapping $App_2, App_3, ...App_8$ gives the score $[30, 35, \cdots 100]$ respectively. The higher distance indicates low similarity and vice versa. A conceptual view of this matrix is illustrated in Figure 4.6.

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
A_1	0	30	35	100	100	70	100	100
A_2	30	0	10	100	80	100	100	100
A_3	35	10	0	100	100	100	100	100
A_4	100	100	100	0	5	100	100	100
A_5	100	80	100	5	0	3	100	100
A_6	70	100	100	100	3	0	100	100
A_7	100	100	100	100	100	100	0	100
A_8	100	100	100	100	100	100	100	0

Table 4.2: Distance-Matrix in initial state.

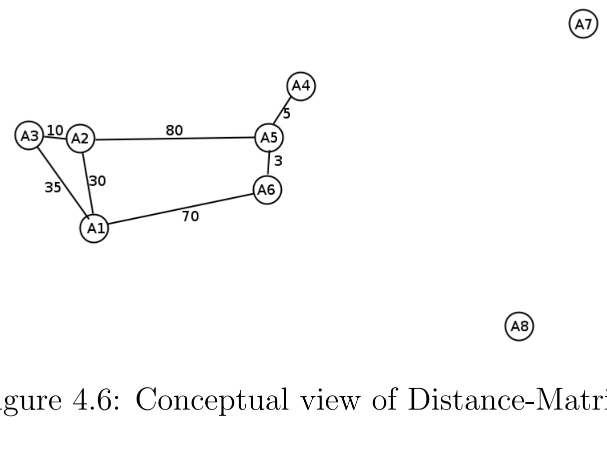


Figure 4.6: Conceptual view of Distance-Matrix.

4.4.2 Clusters: Family Signature

We say two apps are similar only if there is a significant overlap of robust statistical features, i.e. similarity score of those apps is greater than a desired experimental similarity threshold. For example, if the threshold is 35, maximum distance D_{max} between apps within a cluster would be $100 - 35 = 65$.

Using distance matrix, we compute 'neighborhood count' N_c for each app by counting how many of its neighbors are within D_{max}. In every iteration, we consider an app with maximum neighborhood count $max(N_c)$ as cluster center and its neighbors occupying the same cluster. Thus, a large cluster is formed in the first iteration. There might be cases where there would be same neighborhood count for multiple apps. In that case, we calculate the total cost of neighbors T_{cost}, which is the sum of distances to its neighbors, and the app having smallest T_{cost} would qualify for cluster center.

Table 4.3 illustrates the state after a single iteration. A_5 has highest N_c and lowest T_{cost}. A_5 with neighbors A_4 and A_6 form a cluster with A_5 being the representative of the cluster. Thus, the signature can detect all the apps within the cluster as illustrated in Figure 4.7. Once the cluster is formed, we remove the respective points from the distance matrix also as shown in Table 4.4.

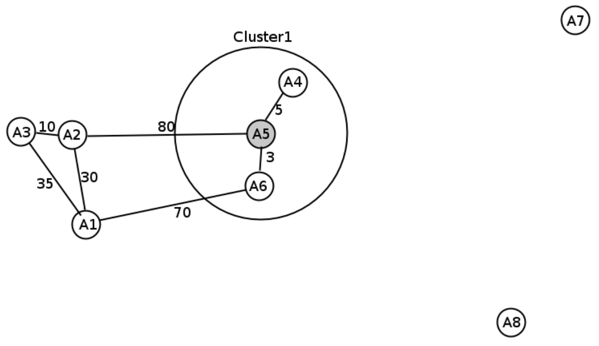


Figure 4.7: Forming cluster with A_5 as representative.

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8	N_c	T_{cost}
A_1	0	30	35	100	100	70	100	100	2	65
A_2	30	0	10	100	80	100	100	100	2	40
A_3	35	10	0	100	100	100	100	100	2	45
A_4	100	100	100	0	5	100	100	100	1	5
A_5	100	80	100	5	0	3	100	100	2	8
A_6	70	100	100	100	3	0	100	100	1	3
A_7	100	100	100	100	100	100	0	100	0	-
A_8	100	100	100	100	100	100	100	0	0	-

Table 4.3: Distance-Matrix after one iteration.

	A_1	A_2	A_3	A_7	A_8	N_c	T_{cost}
A_1	0	30	35	100	100	2	65
A_2	30	0	10	100	100	2	40
A_3	35	10	0	100	100	2	45
A_7	100	100	100	0	100	0	-
A_8	100	100	100	100	0	0	-

Table 4.4: Distance-Matrix after forming cluster A5.

After second iteration of this process, new cluster A_1, A_2, A_3 is formed with A_2 as representative point as shown in Figure 4.8 and its corresponding distance matrix in Table 4.5.

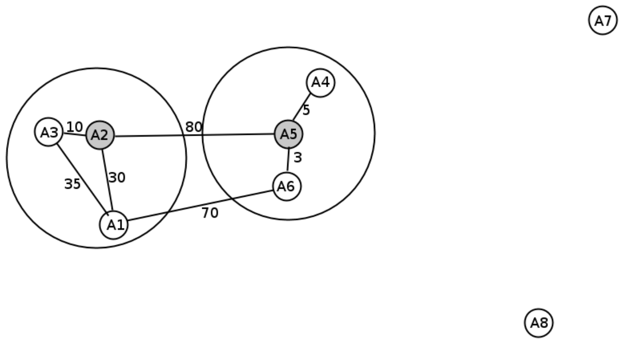


Figure 4.8: Forming cluster with A_2 as representative point.

	A_7	A_8	N_c	T_{cost}
A_7	0	100	0	-
A_8	100	0	0	-

Table 4.5: Distance-Matrix after forming cluster A2.

The clustering process is terminated when there are no neighbors of any point. The apps with entry in the distance matrix are the ones which do not belong to any cluster. These apps are placed in new clusters as shown in Figure 4.9. We are left with the reduced set of signatures $\{A_2, A_5, A_7, A_8\}$, instead of complete set $\{A_1, \cdots A_8\}$. This signature is sufficient for detecting variants of a malware family.

Algorithm 2 illustrates the Signature set reduction. The input to the algorithm is the existing malware signature database, variant signatures and the minimum similarity threshold. This minimum threshold is an experimentally evaluated value. The algorithm generates the reduced signature database removing the unimportant variant signatures to identify the variants with a reduced number of signatures.

Initially, we find the neighbor apps to generate a cluster within the family. This procedure eliminates the signatures with minimum similarity. The apps with signature matching beyond the threshold are clustered together. If maximum number of clusters is reached, then we insert the signature in the distance matrix.

Further, we find the cost of the cluster within the family and repeat this process for signature reduction.

Algorithm 2: Signature Reduction Algorithm.

```
Input  : mal_sign_db – Signature database.
         intra_fam_match(App_i) – Intra-family signature
         match results for every app-i against other apps
         in the same family.
         threshold – Similarity threshold.
Output: redu_sign_db – Reduced signature-set.
1  mal_fam_set ← set of all malware families to which intra_fam_match(App_i)
   belong;
2  foreach family in mal_fam_set do
3      repeat
4          $N_c$ ← An empty array for holding number of neighbor apps;
5          $T_{cost}$ ← An empty array for holding sum of distances to neighbor apps;
6          cl_core_pts ← = Points that are at the cores of clusters;
7          rep_point ← Representative point for the cluster;
8          new_cl_pts ← Points of newly formed cluster;
9          dist_mat ← create_distance_matrix(family);
10         foreach app in family do
11             $N_c$[app] ← find_neighbor_counts(dist_mat, app);
12         end
13         if max($N_c$) == 0 then
14             Insert signatures of all apps in dist_mat to redu_sign_db;
15             break;
16         end
17         foreach app in fam do
18             find_total_costs(dist_mat, app);
19         end
20         cl_core_pts ← find_apps_with_highest_neighbor_count($N_c$);
21         rep_point ← find_apps_with_least_total_count(dist_mat, cl_core_pts,
           threshold);
22         Insert signature of rep_point to redu_sign_db;
23         new_cl_pts ← find_neighbors_of(rep_point, threshold);
24         Append rep_point to new_cl_pts;
25         Remove new_cl_pts from dist_mat;
26     until dist_mat is not empty;
27 end
```

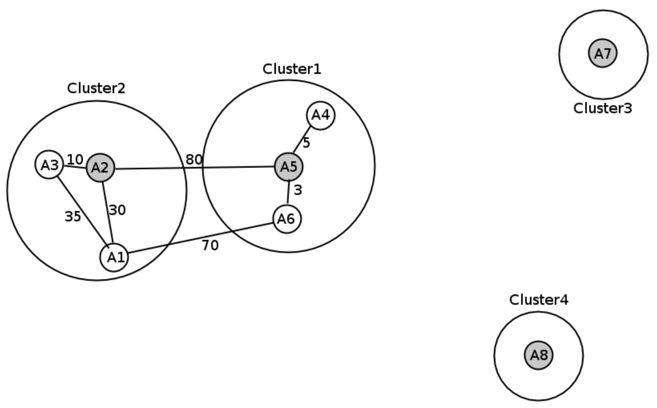


Figure 4.9: Total clusters created after completion of algorithm.

4.5 Experimental Evaluation

The True Positive (TP) and False Positive (FP) rates are performance indices of a malware detector. For the purpose of evaluation, we have collected a dataset of 15,993 Google Play, 3,309 malicious and 5,139 third-party apps summarized in Tables 4.6, 4.10 and 4.7 respectively.

Category	**#Apps**	**Category**	**#Apps**	**Category**	**#Apps**	**Category**	**#Apps**
Arcade & Action	638	Education	600	Music & Audio	535	Sports	491
Books & Reference	593	Entertainment	485	News & Magazines	552	Sports Games	415
Brain & Puzzle	642	Finance	579	Personalization	642	Tools	731
Business	524	Health & Fitness	614	Photography	498	Transportation	519
Cards & Casino	541	Libraries & Demo	435	Productivity	661	Travel & Local	606
Casual	642	Lifestyle	528	Racing	233	Weather	405
Comics	325	Media & Video	596	Shopping	458		
Communication	619	Medical	356	Social	530		

Table 4.6: Category wise Google Play app distribution.

Among the 3309 malicious apps, 1242 belong to 49 families of Malware Genome Project. The effectiveness of the proposed AndroSimilar is tested against obfuscated malicious variants. Following code obfuscation techniques are used to generate transformed malware: 1) Method renaming; 2) Junk method insertion; 3) Control-flow altering; and 4) String Encryption. The experimental *inter-app similarity threshold* is kept 25. If the signature of a particular sample matches with malware signature with similarity score ≥ 25, the sample is identified as malicious. We shall also discuss the implication of changing this threshold later.

Following are the parameter values we considered for experimentation:

App Store	**#App**	**App Store**	**#Apps**
android.d.cn [140]	489	gfan.com [131]	1733
hiapk.com [141]	534	mumayi.com [130]	2363

Table 4.7: Source wise Third party app distribution.

- Feature-size B is 64 and 128 bytes.
- R_{prec} tables generated for APKs with B as 64 and 128. The R_{prec} is also generated for DEX files with B as 64.
- Popularity window size W and $B = 64$.
- R_{pop} threshold value is 16.
- Bloom filter size is 256 bytes.
- The maximum number of features per bloom filter is 160.
- Inter-app similarity threshold is 25.

4.5.1 Byte-sequence length $B = 64$ vs. $B = 128$

The APK experimental evaluation is performed using byte sequence length: (1) $B = 64$; and (2) $B = 128$. We must remember that smaller value of B requires more processing and generates weak features. The larger value of B will generate less features not sufficient for robust signatures [138]. Hence, we have considered $B = 64$. Roussev et al. [138] experimentally evaluated the SDHash and recommend B as 64 or 128 bytes considering the typical network-packet length. Our experimental evaluation suggests that AndroSimilar with $B = 64$ performs better compared to $B = 128$. The normalized entropy distribution for $B = 64$ of both benign and malicious apps is illustrated in Figures 4.3 and 4.4 respectively.

4.5.1.1 Evaluating Unknown Variants

Table 4.8 illustrates the experimental evaluation of unseen malicious apps against 2,854 malware signatures. The training set consists of 2,854 signatures and evaluation set has 455 unseen malware. The test set evaluation is illustrated in the third column of Table 4.8. We can see that the True Positive is 91.43% for the block size $B = 64$. When the apps are evaluated with block size $B = 128$ the result drops by 4% to 86.85.

The same experiment is carried out for the `classes.dex` alone. The sixth column shows TP rate dropping nearly 10% compared to the APK files. DroidMOSS [11] extracted the Dalvik opcode to propose DEX file based signature to identify the repackaged malware. To evaluate the opcode based analysis signature, we extracted the DEX files from the APK archive and trained the signature with Dalvik bytecode. We performed evaluation on 455 `classes.dex` files and obtained 78.25% TP rate.

Category	**Signature Set**	**Test Set**	**% TP APKs 64 byte features**	**% TP APKs 128 byte features**	**% TP DEX 64 byte features**
Unknown Malware	**2854**	**455**	**91.48**	**86.85**	**78.25**

Table 4.8: Experimental evaluation of unseen malware.

4.5.1.2 Evaluating Obfuscated Malware

classes.dex is the app executable file. To implement the code obfuscation, the executable file `classes.dex` must be modified. In the first stage, we extract *classes.dex* from the benign and malicious APK. We generated AndroSimilar signature for APK archive and DEX file to evaluate malware signature for both the archive and DEX file. Table 4.9 illustrates the detection rate for each the APK file with block size $B = 64$ and 128. As illustrated, the obfuscated APK are identified with TP rate 89.62%, 87.77% for $B = 64$ and 79.35% and 77.92% in case of $B = 128$ against method renaming and junk code insertion. The evaluation against other transformation methods is listed in the Table 4.9.

It is important to note that the analysis of only DEX files reduce the TP rate drastically to 5.23%, which is nearly 90% less than its APKs counterpart as listed in the last column. This evaluation is important, as the existing approaches have experimentally evaluated the DEX bytecode signature as and claimed their efficacy [11].

Category	**Signature Set**	**Test Set**	**% TP APKs 64 byte features**	**% TP APKs 128 byte features**	**% TP DEX 64 byte features**
Code Obfuscation					
Method Renaming	3309	234	89.62	79.35	5.56
Junk Method Insertion	3309	234	87.77	77.92	5.13
GOTO Obfuscation	3309	230	88.17	78.70	6.52
String Encryption	3309	158	89.11	76.42	8.23
All Obfuscations at Once	3309	157	84.34	75.97	0.00
Overall	**3309**	**1013**	**87.802**	**77.67**	**5.23**

Table 4.9: Evaluating Obfuscated malware.

4.5.2 Signature-Set Reduction

We evaluated the proposed custom signature-set reduction algorithm on 1,242 samples from 49 Genome Project malware families [2]. The proposed signature reduction algorithm reduces the signature size up to 42% per family as illustrated in Figure 4.10. The maximum reduction is achieved is 88% for *Gone60* and 87% for the *AnserverBot* family. It is important to note that it is possible to detect all the 1,242 malicious samples using just 519 signatures instead of single signature for each malware. The signature reduction improves the analysis speed to identify new variants.

Malware Family	**#Apps**	**Reduced No. of Signatures**	**Reduction Rate (%)**	**Malware Family**	**#Apps**	**Reduced No. of Signatures**	**Reduction Rate (%)**
GingerMaster	**4**	2	50.00	Geinimi	**69**	63	8.70
ADRD	**22**	14	36.36	GoldDream	**47**	42	10.64
AnserverBot	**187**	23	87.70	Gone60	**9**	1	88.89
Asroot	**8**	6	25.00	GPSSMSSpy	**6**	5	16.67
BaseBridge	**122**	22	81.97	HippoSMS	**4**	3	25.00
BeanBot	**8**	4	50.00	jSMSHider	**16**	12	25.00
Bgserv	**9**	3	66.67	KMin	**52**	10	80.77
CruseWin	**2**	2	0.00	NickySpy	**2**	2	0.00
DroidDream	**16**	14	12.50	Pjapps	**58**	35	39.66
DroidDreamLight	**46**	44	4.35	Plankton	**11**	9	18.18
DroidKungFu1	**34**	11	67.65	RogueLemon	**2**	2	0.00
DroidKungFu2	**30**	14	53.33	RogueSPPush	**9**	2	77.78
DroidKungFu3	**309**	98	68.28	SndApps	**10**	5	50.00
DroidKungFu4	**96**	45	53.13	YZHC	**22**	5	77.27
DroidKungFuSapp	**3**	1	66.67	zHash	**11**	6	45.45
FakePlayer	**6**	4	33.33	Zsone	**12**	10	16.67
Others	**2067**	NA	NA				

Table 4.10: Android Malware Genome dataset.

4.5.3 Comparing AndroSimilar

Software similarity is a measurement to detect plagiarism at the file level to identify similar content. Our approach works at file level for extracting robust statistical features to detect repackaged malware, variants of known malicious families and obfuscated malware. DroidMOSS [11] is based on Context Trigger Piecewise Hashing (CTPH) [142], a fuzzy hashing technique used to identify spam emails. DroidMOSS generates an 80-byte signature from the Dalvik opcode to identify repackaged malware. Suspected app features are verified against the original apps using the edit-distance algorithm to identify the similarity score.

The proposed AndroSimilar generates normalized entropy features such that they rarely occurs among unrelated files. We have experimentally evaluated the DroidMOSS signature on the AndroSimilar database to assess the performance. Conclusively, Vassil Roussev found that Similarity Digest Hashing (SDHash) outperforms SSDEEP with precision rates of 94% and 68% respectively [139]. Due to the fixed-length signature, SSDEEP performance depends on the file size. The SSDEEP algorithm can reliably co-relate that file size upto 3 MB. The SDHash has the upper limit of 1GB for the large file. The experimental evaluation of AndroSimilar which is based on SDHash outperforms DroidMOSS.

Section 4.5.3 discusses the advantage of AndroSimilar. DroidMOSS considers `classes.dex` opcode to detect repackaged malware. However, AndroSimilar considers the complete APK file as the `androidmanifest.xml`, assets and APK resources are easy targets for repackaging the malicious apps. Furthermore, DroidMOSS can be easily evaded by simple code obfuscation techniques as the Dalvik methods and classes are rearranged during the transformation. The repackaged apps have almost the same resources, but bytecode is modified by obfuscation to evade the DroidMOSS signature. The opcode sequence does not consist high-level semantic information contributing towards false negatives.

DroidAnalytics [72] proposed a static analysis framework for effective detection of obfuscated Android malware. Signature generation is done at method, class and app level in sequence to detect obfuscated or repackaged malware. The approach is effective against simple obfuscation techniques like method renaming, control flow goto-obfuscation, and string encryption. The proposed framework for static analysis does not perform detailed app analysis.

The DroidAnalytics considers Dalvik “Opcode” targeting the DEX files. We have evaluated the DroidAnalytics against the experimental dataset for comparison with AndroSimilar. The proposed AndroSimilar outperforms DroidAnalytics approach in identifying repackaged and obfuscated malware. The DroidAnalytics is more suitable for general purpose malware detection. The proposed AndroSimilar is suitable for identifying a particular class of malicious apps like repackaged malware, variants of existing malicious apps and code transformed malware. Furthermore, DroidAnalytics approach does not report correct malicious app detection or inaccuracy in the signature extraction methodology.

RiskRanker [28] is a multi-step heuristics signature technique to analyze apps exhibiting dangerous behavior. The method identifies class path as a malware feature to detect multiple code mutations. The Riskranker approach is a two phased system to: (1) identify known malicious payload by storing their signature and use heuristics for the variant of existing malware; (2) identify practices rarely observed among legitimate apps like unsafe bytecode and native code loading. However, the heuristics can be evaded by changing the opcode pattern with execution order. The proposed AndroSimilar is a file feature extraction technique for generating robust statistical features using a novel signature algorithm.

4.6 Discussions

In this Section, we discuss and evaluate the reasons for selecting the complete app for signature instead of the Dalvik bytecode considered by other analysis approaches.

4.6.1 APK Signature vs DEX Signature

classes.dex in the APK archive stores Dalvik executable bytecode. It is small size compared to an APK. If only DEX file can generate statistically robust features, the efficiency of scanning and detection can improve further. Fuzzy-hashing based approach [11] evaluated the Dalvik-opcodes within DEX files to detect unseen malware and repackaged apps. Following the insight, we evaluated AndroSimilar with DEX input. R_{prec} table is generated from the entropy distributions of benign and malicious DEX illustrated in Figure 4.10 for 1,500 benign and 1,242 malware respectively.

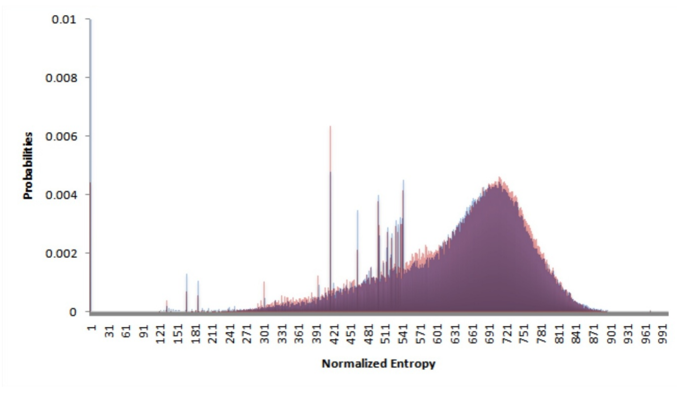


Figure 4.10: Normalized DEX file entropy distribution.

We can see negligible concentration at the higher entropy end; weak features are removed at the lower entropy end. AndroSimilar generates signatures for apps to identify unseen variant and obfuscated malware. We have observed empirical distribution and results for both 64 and 128 byte-length features. Based on experimental evaluation as suggested in [138], 64-byte features gives reasonable detection rate. We analyzed AndroSimilar *classes.dex* executable bytecode. However, considering the DEX files reduces the detection rate. The repackaged applications modifies the manifest, assets, resources, pictures or dynamically load executable code. Considering only Dalvik bytecode reduces the possibility of identifying the robust features. It also gives low detection rate for repackaged variants, contrary to fuzzy-hashing based approach [11] used to detect repackaged apps.

4.6.2 Comparing the Block size

Experimental evaluation demonstrates AndroSimilar's effectiveness in detecting zero-day malware. The Signature generation is automated to scale the increased unique malware instances. We checked samples collected from Google Play and third-party app stores against the signature set of 3309 malware: $B = 64$ and $B = 128$. The results illustrated in Table 4.11 report 157 Google Play apps and 152 third-party apps identified malicious.

The manual analysis identifies them as benign. 43 Google Play and 128 third-party apps were reported original versions of their malicious counterparts. Hence, we concluded that malware authors reverse-engineered the original apps from app stores, added malicious payloads and repackaged them. The remaining samples detected malicious with AndroSimilar signature were false-positive (Google Play: 0.71%, Third-party market apps: 0.47%). Table 4.12 lists unseen repackaged malware identified malicious with AndroSimilar signature. The proposed AndroSimilar signature identifies the third party app used for malicious repackaging. Hence, the proposed AndroSimilar can be used to identify cloned and repackaged apps.

Category	**Signature Set**	**Test Set**	**% FP APKs 64 byte features**	**% FP APKs 128 byte features**
Google Play	3309	15993	0.98	0.78
Third-Party	3309	5139	2.96	2.30

Table 4.11: Detection Results of Google Play and Third-Party apps.

Signature size of an app is also an important factor due to the storage constraints of a smartphone. Signature size depends on R_{pop}, the popularity threshold retaining rare features. Higher the threshold, less the number of features retained [138], reducing the signature size. Furthermore, the byte-sequence length B also controls the signature size. Higher the value of B, lower the granularity [138], reducing the signature size. We performed experimental evaluation based on the SDHash algorithm [138] with R_{pop} 16 as threshold. Experimental evaluation on a larger dataset demonstrates that Block size $B = 64$ gives better detection rate with reasonable performance.

Evaluation of AndroSimilar for DEX files reduces the detection rate for obfuscated app (TP rate less than 4%). However, considering the complete APK, the same obfuscated malware, TP rate is beyond 90%. The reasons for ineffectiveness is the inability to identify modifications among resources, assets, and manifest. The *inter-app* similarity threshold is experimentally selected value 25 to balance the false alarm rate (FP and FN) [137]. The similarity threshold can be set according to the analysis system necessity. Table 4.12 lists unseen malware variants identified as malicious apps from the third party market apps. The above malware samples were detected benign by the commercial anti-malware and existing signature based analysis techniques. Hence, AndroSimilar can be employed as a malware detector at online app markets.

Third Party App	**Malicious App**	**Comment**
SHA-1:2ea5f6dc465cf89caab438ae8bcb5b42de3e444f MD-5: ee8be1b7f2761f42957bb22b2a0f6414 Package Name: com.requiem.armoredStrike	SHA-1: bdd581d2d57ad71b0bf6401e76c8120cd5dc0173 MD5: 5d27c7d0c5630f4c7a8b7a8f45512f09 Package Name:com.requiem.armoredStrike **Malware Family : Geinimi**	**Disguised as updated version** Original Main launcher receiver changed **Dangerous permissions added:** ACCESS_FINE_LOCATION, ACCESS_GPS, CALL_PHONE, READ/WRITE CONTACTS READ/SEND_SMS, READ/WRITE HISTORY_BOOKMARKS.
SHA-1: fc438c6d0cabc2190c26b41e9dc5d3d3843274eb MD5: ec3b45dc3ebda87cc420722f1895e75c Package Name: com.camelgames.hyperjump	SHA-1: 00983aad12700be0a440296c6173b18a829e9369 MD5: 513971a8cde07e145a85a8707f83e4b5 Package Name: com.camelgames.hyperjump **Malware Family : Pjapps**	**Service added:**Intent receivers for SMS_RECEIVED, WAP_PUSH_RECEIVED. **Dangerous permissions added:** RECEIVE_SMS, RECEIVE_MMS, SEND_SMS, NEW_OUTGOING_CALL, READ/WRITE_HISTORY_BOOKMARKS, INSTALL_PACKAGES.
SHA-1: 97bc745f247da451995a0be635150eb71a3c0f2f MD5: 07e640792506d889b67e4a7061a9aff7 Package Name: jp.hudson.android.militarymadness	SHA-1: 1f522d9ab07fc716e7f201fad12ccea396987a83 MD5: 6ea15fdda8ba208b19e5c7131e9d413f Package Name: jp.hudson.android.militarymadnes **Malware Family: GoldDream**	**Very minor change in package name and activities added.** Intent receiver for BOOT_COMPLETED, SMS_RECEIVE, PHONE_STATE, NEW_OUTGOING_CALL added. **Dangerous Permissions added:** RECEIVE_SMS, READ_SMS, INSTALL_PACKAGES, RECEIVE_BOOT_COMPLETED, HISTORY_BOOKMARKS.
SHA-1: 2b11e7d3bc8421da143deab57acaea03e0a83b1c MD5: 21d81b064951e9ed7beff98d6879e55a Package Name: jpcom.wuxi.GoldMiner.domob	SHA-1: b9d992b88ef1a4a75362f8f5d069716ea7a3321e MD5: 025a55c1bcbd3be2ca03aa314ce9a4c2 Package Name: jpcom.handcn.GoldMiner.free **Family: Geinimi**	**Original Main launcher receiver changed.** Advertisement Publisher ID changed, along with the ad-library **Dangerous permissions added:** CALL_PHONE, INSTALL_SHORTCUT, READ/WRITE_CONTACTS, SEND/READ_SMS, READ/WRITE_HISTORY_BOOKMARKS, ACCESS_GPS, ACCESS_LOCATION.
SHA-1: 9ef6fc25d9e599ce6bfa7c3fb4d21a775f5cf98f MD5: c9caebc6cb727d720e04cb77ce1e042e Package Name: com.camelgames.mxmotorlite	SHA-1: ef140ab1ad04bd9e52c8c5f2fb6440f3a9ebe8ea MD5: e5b7b76bd7154dea167f108daa0488fc Package Name: com.camelgames.mxmotor **Family: AnserverBot**	**Many new activities and services added.** Receiver for SMS_RECEIVED, BOOT_COMPLETED, PICK_WIFI_WORK added. **Dangerous Permissions added:**READ/RECEIVE/WRITE_SMS, RECEIVE_BOOT_COMPLETED, READ/WRITE_CONTACTS, CALL_PHONE, READ_PHONE_STATE.
SHA-1: 218af28eeb168dd16df6d0faacb3f77f51fc66df MD5: 4b0c93b1a14b5fdad60715a8a6b1987e Package Name: com.moregame.drakula	SHA-1: b9891ab782cf643c81f0f1c130ea119384dbefe1 MD5: 8498984d8f9b7260fd032d6f0a2534aa Package Name: com.moregame.drakula **Family: Geinimi**	**Original Main launcher receiver changed.** **Receiver for BOOT_COMPLETED event added.** **Dangerous permissions added:** ACCESS_FINE_LOCATION, CALL_PHONE, READ/WRITE_CONTACTS, READ/SEND_SMS, READ/WRITE_HISTORY_BOOKMARKS, ACCESS_GPS, ACCESS_LOCATION added.

Table 4.12: Third Party apps detected malicious by AndroSimilar.

4.7 Selecting Signature Threshold

The proposed AndroSimilar experimental similarity threshold is set to 25 as optimal value for unseen variant detection. The value has been determined empirically after experimenting a range of threshold (20-47) to achieve high TP and low FP. The SDHash algorithm correctly detect the malicious variant with zero FN; hence, malicious app cannot circumvent the AndroSimilar signature.

We performed empirical evaluation of different threshold on 3309 malware and 2732 Google Play apps. The threshold value 25 gives optimal TP and reduced FP [137] during the real test. Figure 4.11 illustrates the results of experimental evaluation for different threshold values.

Figure 4.11 is a comparison of TP and FP values to justify empirical threshold value. As the similarity threshold increases, FP reduces at the cost of TP. Hence, we have selected 25 as signature similarity threshold. However, one can modify the threshold value as per the requirement of the detector.

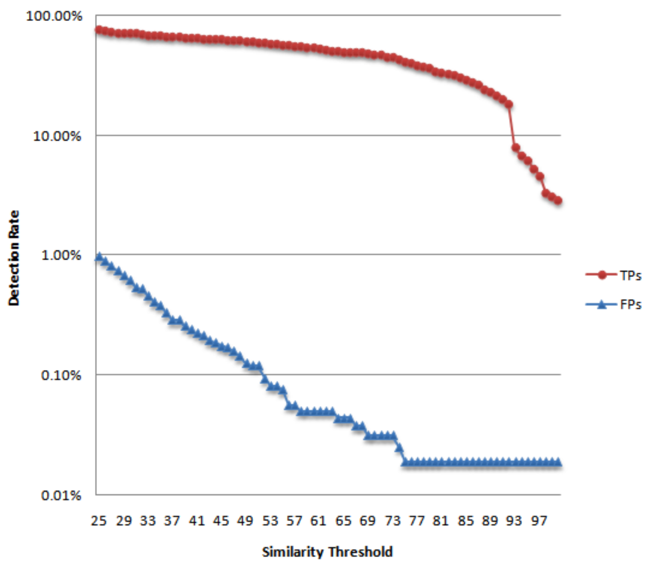


Figure 4.11: Effect of similarity threshold on True-Positives and False-Positives.

4.8 Summary

In this chapter we proposed and evaluated AndroSimilar to detect variants of known Android malware families, obfuscated malware and repackaged apps. The proposed method utilizes statistically robust features generated using SDHash algorithm. The proposed technique generates a variable-length signature for the suspect APK file. AndroSimilar effectively detects known malware and variants of existing malware families with an True Positive $> 90\%$.

The proposed signature is resilient to trivial transformation methods like string encryption, method renaming, junk method insertion and control-flow obfuscation. We also report the false-positives generated from Google Play and third-party app stores were the original apps of their malicious counterparts. The AndroSimilar can be employed to detect repackaged apps from app stores. Our evaluation is also consonant with experimental evaluation in [138] suggesting 64-bytes feature size better suited compared to 128-byte feature. We evaluated our custom algorithm to reduce signature-set against 49 malware families from Malware Genome Project [2] and report 42% signature reduction. We also applied AndroSimilar to just DEX files. The analysis evaluation reports reduced detection rate in case of obfuscated malware. We suggest analysis of APK rather than the Dalvik bytecode.

Chapter 5

CONFIDA: COvert Feature Misuse analysis using ICC

In the previous chapter, we discussed AndroSimilar, a signature based detection technique for repackaged malware and variants of malicious apps. The evolving malware employs functionality such as covert SMS sending, dial premium-rate numbers, record audio/video and click pictures without user consent or knowledge. We identify such covert actions as *sensitive feature misuse*. In this chapter, we propose Android ICC based CIG technique to detect covert feature misuse evading the existing anti-malware. The encouraging results on one thousand notable SMS Trojans and Spyware indicates that the proposed technique can be deployed as a static app vetting framework for app markets.

5.1 Android Inter-Component Communication

The proposed approach identifies sensitive functionality necessitating explicit user intervention. Android platform being event-driven, capturing every possible control-flow path is a challenge. The Android components are asynchronous and communicate using ICC. In this chapter, the term "covert malicious" behavior is defined as app activity executed without explicit user consent.

5.1.1 Inter-Component Communication

To enforce secure communication, Android facilitates communication and sharing of data among apps via Inter-component communication API [33]. When a component initiates ICC, the reference monitor looks at the permission labels assigned to its container app. If the target component access permission label matches the said collection, Android framework permits the ICC. In case the label is not a part of the collection, ICC is refused even if the app is signed with the same certificate. The app components interact using *Intent*, a higher level intra and inter app communication abstraction. The developer creates *Intent* object that contains the address of the target component. An *explicit* intent reaches the desired component that claims to carry out an action. In case of multiple targets, user is allowed to select the component of the same priority.

5.1.2 ICC based Component-Interaction Graph (CIG)

The Dalvik bytecode reverse engineering is easy due to the presence of high level semantic information such as type and name of variables, fields and methods. Hence, the proposed technique analyzes Dalvik bytecode to identify covert behavior using Component Interaction Graph. The CIG is a directed graph, where a node represents a sequence of Dalvik bytecode instruction block. The APK is represented as a graph G(V, E) where V represents a node and edge E is the path dependency between the two nodes. The CIG edges illustrated in Figure 5.1 are represented as:

- *Conditional edge* (`if, for, etc.`) from one node to another within a method.
- *Synchronous edge* represents a direct flow from one node in method m_1 to another node in method m_2 of a graph. The method invocation in Java is an example of synchronous flow.
- *Asynchronous edge* represents an indirect flow from a node in method m_1 to another node of method m_2 in a graph. For example, forking a `java.lang.Thread` or scheduling a `java.util.TimerTask` is an asynchronous flow.

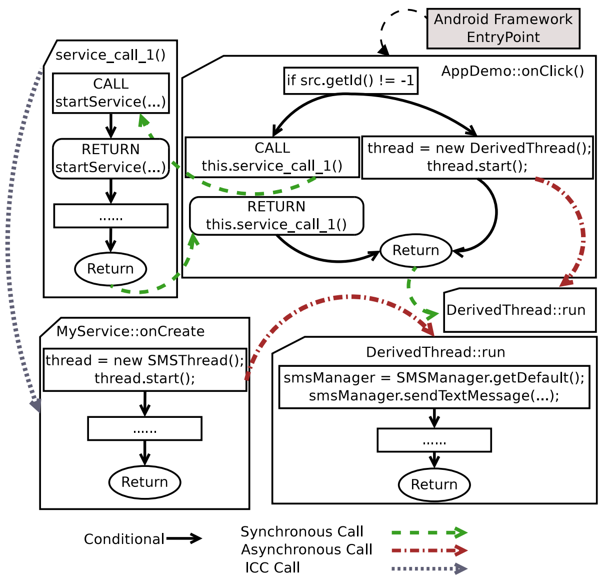


Figure 5.1: ICC Control flow.

- *ICC edge* represents the interaction between different components of an app. For example, an *activity* component interacts with a *service* component using `startService()` API call.

As illustrated in the Figure 5.1, the CIG construction starts with the Android framework entry point. The nodes representing the code block are connected with synchronous, asynchronous, and ICC edges. The schematic representation in Figure 5.1 illustrates different line styles to indicate the communication mode (synchronous, asynchronous, or ICC). When a method is invoked, it is represented by the `solid line`. A dotted line represents an indirect, asynchronous control flow between the two methods. The dashed edges illustrates the interaction between components.

5.1.3 Motivation for proposed CONFIDA

In this Section, we illustrate a motivating example of a real world Android malware FakeInstaller in Listing 5.1. It is one of the top ten malware employing extensive obfuscation evading the existing approaches [94, 95, 86, 84, 143]. As illustrated, the line number 5 checks for the presence of emulator used for app analysis or development. Presence of such environment impede the real device infection. In line number 9, the class and method names are obfuscated to erase the program semantics. For example, a random string value “VQIf3AInVTTnSaQI+R]KR9aR9”, is decrypted to `android.telephony.SmsManager` class loaded via reflection API to send premium-rate SMS without explicit user consent.

This is visible in the line number 26, as the string "BaRIta*9caBBV]a" is decrypted to the `SendTextMessage` method. Furthermore, in line number 25, `getMethod` sends SMS using text from the parameters declared in line number 1. Hence, the most relevant work proposed in [94, 95] fail to address such complex behavior. Furthermore, AsDroid [95] can be easily defeated by replacing the text UI with an image or icon as discussed in [144].

```
public static boolean gdadfjrj                                        1
(String paramString1,String paramString2){ [...]                     2
                                                                      3
// Anti analysis check to evade emulator                              4
if (zhfdghfdgd()) return;                                             5
                                                                      6
// Get class instance                                                 7
Class clz = Class.                                                    8
forName(gdadfjrj.gdafbj("VQIf3AInVTTnSaQI+R]KR9aR9"));                9
Object localObject = clz.getMethod(gdadfjrj.                         10
gdadfjrj("]a9maFVM.9"), newClass[0])                                 11
.invoke(null, new Object[0]);                                        12
                                                                     13
// Get the method name                                               14
String s = gdadfjrj.gdadfjrj("BaRIta*9caBBV]a");                     15
                                                                     16
// Build parameter list                                              17
Class c = Class.forName(gdadfjrj.                                    18
gdadfjrj("VQIf3AInVTTnSaQI+R]KR9aR9"));                              19
                                                                     20
Class[] arr = new Class[]                                            21
{nglpsq.cbhgc, nglpsq.cbhgc, nglpsq.cbhgc, c, c };                   22
                                                                     23
// Reflection for invoking the method to send SMS                    24
clz.getMethod(s, arr).invoke(localObject, newObject[]                25
{ paramString1, null, paramString2, null,null });                    26
```

Listing 5.1: Motivation for CONFIDA [145, 31]

We consider the invocation of sensitive resource without user consent as a trigger for deviation from the benign behavior. To illustrate this, Figure 5.2 compares the behavior of a normal SMS app with HippoSMS, a premium-rate SMS Trojan subscribing the premium SMS service without explicit user consent. Figure 5.2 (i) sends `sendTextMessage()` upon receiving the user input through user interface. Figure 5.2 (ii) illustrates SMS sending without any user approval, a typical SMS Trojan behavior. The app subscribes premium SMS service to a hard-coded telephone number. The `sendSMS()` method is invoked without user-interaction with a pre-defined number in the string `s0`. Similar covert behavior such as call recording, audio/video recording and picture click are leveraged by the evolving Android malware [113]. The malware authors are heavily employing environment detection techniques and using transformation methods to evade the analyzer. Furthermore, the employed tricks provides the malware more infection and propagation time.

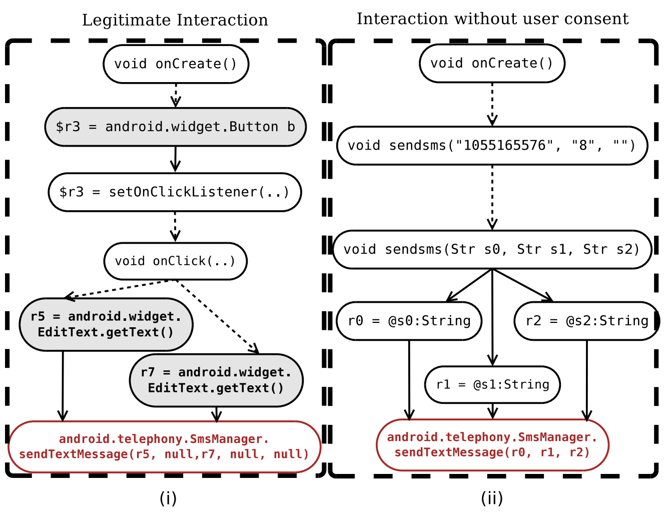

Legitimate Interaction
Interaction without user consent
void onCreate()
$r3 = android.widget.Button b
$r3 = setOnClickListener(..)
void onClick(..)
r5 = android.widget.EditText.getText()
r7 = android.widget.EditText.getText()
android.telephony.SmsManager.sendTextMessage(r5, null,r7, null, null)
void onCreate()
void sendsms("1055165576", "8", "")
void sendsms(Str s0, Str s1, Str s2)
r0 = @s0:String
r2 = @s2:String
r1 = @s1:String
android.telephony.SmsManager.sendTextMessage(r0, r1, r2)
(i)
(ii)

Figure 5.2: Illustrating legitimate and feature misuse.

5.2 Proposed CONFIDA

Following is the summary of contributions of CONFIDA:

1. We propose to identify Android apps employing covert malicious behaviors such as sending SMS, dial premium rate numbers, audio/video recording and clicking pictures without explicit user consent. We generate a component-interaction graph to identify the covert feature misuse.

2. We design a novel automated approach CONFIDA, a static analysis technique for precise CIG based analysis, considering the ICC and asynchronous Android API. The component-interaction-graph is an ICC based graph leveraging data flow analysis and asynchronous Android API to identify covert feature misuse.

3. CONFIDA performs interprocedural Dalvik bytecode data-flow analysis to generate CIG. Furthermore, the proposed methodology considers the asynchronous Android API such as `java.lang.Thread` and AsyncTask callback `onPreExecute, onPostExecute and onProgressUpdate` methods.

Table 5.1: Entrypoints with corresponding API.

Android Component	**Entrypoint**	**Corresponding API**
Activity	onCreate, onStart, onPause	startActivity, startActivityForResult, onActivityResult
Service	onCreate, onBind, onStartCommand, onDestroy	startService, bindService, stopService
Content Provider	onCreate	query, insert, update
BroadcastReceiver	onReceive	sendBroadcast, sendStickyBroadcast, sendOrderedBroadcast

4. We analyze 1,154 Google Play and 954 real-world malware from Android malware genome [2], contagiodump [127], droidbench [146] and IccRE [84] dataset and identify the feature misuse evading the existing anti-malware.

Here, we justify the use of ICC-based control-flow analysis for sensitive feature extraction. The proposed approach analyzes the sensitive resource invocation listed in Table 5.2 captures the feature misuse invoked without explicit user consent. The Dalvik bytecode is converted systematically into a control-flow graph (CFG), where each node of the graph represents a block of Dalvik instructions. The control-flow analysis identifies the execution semantics of the program. The data-flow analysis is important to understand the program behavior. Hence, we augment the CFG with ICC information using interprocedural data-flow analysis discussed in [147].

Feature	**API(s)**

Sending SMS	android.telephony.SmsManager::sendTextMessage()
	android.telephony.SmsManager::sendDataMessage()
	android.telephony.SmsManager::sendMultipartTextMessage()

Phone Calls	Intent callIntent = new Intent(Intent.ACTION_CALL, Uri.parse(number));
	startActivity(callIntent);

Device Information	android.telephony.TelephonyManager.getDeviceId;
	startActivity(android.telephony.TelephonyManager.getCellLocation);
	android.telephony.TelephonyManager.getSimSerialNumber;
	android.telephony.TelephonyManager.getLine1Number ;

Recording Audio	android.media.MediaRecorder::setAudioSource()
	android.media.MediaRecorder::start()

Recording Video	android.media.MediaRecorder::setVideoSource()
	android.media.MediaRecorder::start()

Taking Pictures	android.hardware.Camera::takePicture()

Table 5.2: Sensitive features and corresponding API used by CONFIDA.

The features of interest that can be misused by the evolving malware are listed in Table 5.2. In particular, we identify `sendTextMessage(), sendDataMessage()` methods to detect premium rate SMS misuse. The premium rate phone number dialing is a user activity. Hence we monitor the startActivity to identify misuse of call feature. Similarly, `setAudioSource()`, `setVideoSource()`, `takePicture()` and `start()` methods are monitored during the analysis.

5.2.1 Implementation Details

A normal application performs sensitive operations such as sending SMS, phone call, audio/video recording, call recording, picture click with explicit permission from the user. However, the advanced malware SMS Trojans and spyware covertly abuse the sensitive features bypassing the most significant user consent trigger, a prime necessity to activate the sensitive functionality. If our hypothesis holds, the control-flow dependence analysis between the user initiated action and a particular operation can identify the anomaly from benign behavior. For example, the user action can be (i) click send button for SMS sending; (ii) provide input text; or (iii) click a button to record the audio. We consider the sensitive API calls on user interface returning the user input as explicit consent and interaction triggers listed in Table 5.2. Figure 5.3 illustrates steps to detect covert feature misuse.

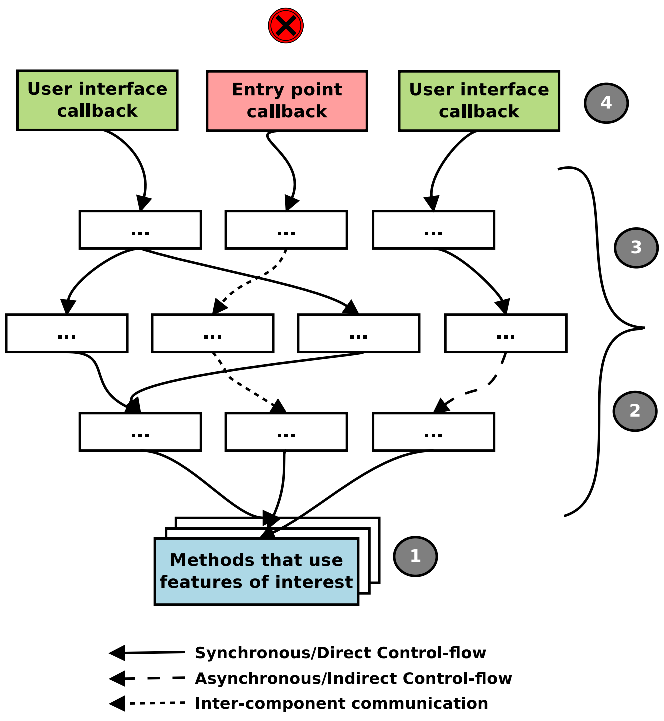

User interface callback
Entry point callback
User interface callback
4
3
2
1
...
Methods that use features of interest
Synchronous/Direct Control-flow
Asynchronous/Indirect Control-flow
Inter-component communication

Figure 5.3: CONFIDA approach: (1) Identify features of interest. (2) generate ICC based CIG. (3) ICC and data dependence analysis. (4) Reverse reachability analysis (UI or callback entry point).

The steps given below gives an overview of the analysis procedure:

1. Disassemble the APK file to extract the `Androidmanifest.xml` and Dalvik bytecode from `classes.dex`. Identify the app components declared in the manifest file.
2. Identify the methods (features of interest) that invoke sensitive functionality within the Dalvik bytecode. The features of interest are listed in Table 5.2.
3. Build a precise and complete control-flow considering the asynchronous nature of Android API and the ICC.
4. Perform control-flow dependence analysis from the Dalvik bytecode considering the inter- component communication. Perform reverse path reachability in the CIG toward the top-level methods. Identify the user initiated input trigger.
5. Classify the *top-level* methods as: *user triggered* or *framework entry point callbacks.*
6. If the entry point callback is received, the application has a covert access to the sensitive functionality.

In the first step, we decompile the Dalvik bytecode and generate programmable structures including interprocedural control-flow graph. Initially we construct the graph by splitting the Dalvik instructions of the methods into basic blocks. The basic block consists of instructions having a single entry and exit point respectively. The Java exceptions present in the Dalvik bytecode must be considered properly as the exceptions blocks do not have parents. Once we build the CFG, we have to perform the data-flow analysis. The data-flow based analysis converts the Dalvik bytecode order to propagate information between the CFG blocks. Hence, we augment this control-flow with asynchronous information.

In the next step, we perform the inter-procedural data-flow analysis to identify the Android system callbacks. More precisely, we consider `Thread`, `Runnable`, `TimerTask`, `CountDownTimer`, `AsyncTask` and `Handler` API calls. As illustrated in the Figure 5.3, we further augment the control-flow with the ICC information. We build the app CIG considering the interprocedural data-flow analysis [147] on the sensitive API listed in Table 5.2. The node of the CIG represents a component, whereas edge from one node n_1 to other n_2 represent the fact that component n_1 launches n_2.

Initially, we search for the source components within the Dalvik bytecode that use ICC API calls. Once the feature of interest is identified, we analyze the arguments using data-flow analysis. The constant argument values are further analyzed to identify the potential target components launched from the source. As the *content providers* produce data related services, we exclude its information while building the CIG. The class hierarchy analysis resolves the *virtual* functions, an important parameter to generate a precise control-flow. For example, a method `start()` is invoked using an object type `Thread`. This method would resolve to any class derived from the `Thread`. Then we identify the method callback toward the top-level methods.

5.2.2 Augmenting the Control-Flow

We augment the control-flow by (i) invoking implicit methods and (ii) ICC using the Android Intents. The proposed approach performs control-flow analysis to identify framework callback. The idea is to determine the reachability of the user initiated triggers. The intent based control-flow analysis captures the dependence relation between the apps and its defined components. An Intent can create a new activity (`startActivity, startActivityForResult`), methods and communicates between the components. Explicit Intent gets delivered to the desired component. The implicit Intent is delivered to any component performing the required operation.

For an explicit Intent, the target component is already known. Hence, we identify the source and destination components. The target component is linked to the intent to capture the component control-flow. The intent object constructor is analyzed to extract the provided target component. In case it is not given, we find the parameters from the methods of intent object (`setClass(), setComponent(), setAction()`) to identify the target components. This information is used to augment the graph by identifying the source and target components for all explicit Intents.

For an implicit `Intent`, a component that declares its ability to handle a method becomes the target component. Android system determines existence of the target component by specifying inside the `Androidmanifest.xml`. The proposed approach handles implicit intents by extracting the components from Android manifest and actions associated with them to identify target component. Implicit invocation is communicated to the framework by using `registration` method. For example, `onClick` callback employs `setOnClickListener` method to bind with the UI. When a user taps the button, Android framework invokes the `onClick`. The proposed approach considers the Android event handlers to augment the graph with data flow information. However, encrypted code, reflections, dynamic code loading are out of scope for static analysis and hence necessitate dynamic analysis functionality.

5.2.3 CIG Reachability

The reachability analysis identifies the unreachable code that (class/method) is never executed. Hence, we construct an inter and intra control-flow analysis to identify the execution path possibilities. In particular, we move up toward the entry point through the top level methods listed in Table 5.2 to identify the user-initiated trigger and corresponding API reachable from the main activity.

5.3 Evaluating CONFIDA

In this Section, we present the evaluation of proposed CONFIDA. The idea is to create a complete and precise control-flow graph using ICC to detect covert malicious behaviors missed by the existing methods. We evaluate the effectiveness of CONFIDA by analyzing some notable and notorious SMS Trojans and Spyware apps. Moreover, we take benign representative apps from Google Play to ascertain the legitimate usage of sensitive features and report the false positives.

5.3.1 Experimental Setup

Experiments are performed on Intel Core i7 machine with 8 GB RAM. We evaluate the proposed approach against: (i) Droidbench testsuite [146]; (ii) known malware apps; (iii) Google Play apps; and (3) Obfuscated malware. We conducted the first experiment on 1,260 malware genome [2], 207 VirusShare repository [128] and 35 contagiominidump [127] malware.

5.3.2 Evaluating known Malware

We conducted the first experiment on 1,260 malware from Android Genome [2], 207 malware from VirusShare [128] and 35 contagiominidump [127]. The discussed repositories have real Android malware app instances collected from various sources including third party regional app stores, the likely malware sources [140, 141, 131]. Table 5.3 enlists the package name, sensitive features, number of analyzed samples and number of execution paths triggered without explicit user approval.

Package Name	**Sensitive Feature(s)**	**# of Sensitive Feature Paths**		
		Total	**Without User Triggered Events**	**Correctly Detected by CONFIDA**
Malware Apps				
com.ku6.android.videobrowser (HippoSMS)	SMS	2	2	2 ✔
org.me.androidapplication1(FakePlayer)	SMS	1	1	1 ✔
kagegames.apps.DWBeta (Dog Wars)	SMS	1	1	1 ✔
t4t.power.management (GGTracker)	SMS	1	1	1 ✔
com.talkweb.ycya (RogueSPPush)	SMS	1	1	1 ✔
com.mobile.app.writer.zhongguoyang (Pjapps)	SMS, Audio	3	3	3 ✔
com.software.application (SMS Boxer)	SMS	7	3	3 ✔
com.parental.control.v4 (Dendroid)	SMS, Call, Audio, Video, Photo	6	6	6 ✔

Table 5.3: Evaluating known malware.

As illustrated in Table 5.3 illustrates notable malware apps employing covert feature misuse. The HippoSMS `com.ku6.android.videobrowser` invokes two sensitive feature paths. The proposed CIG identifies both the paths invoked without user triggers. Similarly Dendroid malware (`com.parental.control.v4`) has SMS, call, audio, video and Picture click misuse paths. The CONFIDA detect all 6 feature misuse paths invoked without user consent. The other known malicious with feature misuse are listed with corresponding details in the Table given above.

Analyzing the Google Play app `com.wn.message` and `com.me.phonespy` illustrated in Table 5.4, we find sensitive features invoked without user consent. `superdial` `and callrecorder` invoke the sensitive functionality to dial a number to automatically record Phone call. If a user approves the installation and gives explicit consent, these apps generate false positives.

Package Name	**Sensitive Feature(s)**	**# of Sensitive Feature Paths**		
		Total	**Without User Triggered Events**	**Correctly Detected by CONFIDA**
Benign Apps				
cn.menue.superredial[1]	Call	2	1	2 ✔
com.wn.message	SMS	3	0	0 ✔
polis.app.callrecorder[1]	Audio	1	1	1 ✗
com.me.phonespy[1]	Photo	1	1	1 ✔
com.Rainbow.hiddencameras	Photo	1	0	0 ✔

Table 5.4: Evaluating Google Play apps.

The Phonespy app (`com.me.phonespy`) has a picture misuse path apart from the call recording feature. This Google Play app has no reason to have hidden picture click capability. Similarly, `com.rainbow.hiddencameras` package also has a picture click misuse path correctly identified with CONFIDA. The `polis.app.callrecorder` records the call and writes to the external storage. This facility is already defined in the features.

Evaluation results of CONFIDA on a dataset of 2108 apps (951 malware and 1,137 benign) is illustrated in Table 5.5. The benign apps include popular apps such as *Whatsapp*, *Viber*, *True Caller* and spying apps (i.e., *Hidden Camera* and *Call Recorder*) from the Google Play. CONFIDA generated 11 false positives. However, 10 of the 11 apps are spying apps where the user explicitly consents to a task that does not require intervention.

To evaluate the existing malware, we experimented a larger dataset consisting benign and malicious apps listed in Table 5.5. The Table lists family name, number of samples analyzed for the particular family with false negative and false positives for the family. False negative is generated when the proposed approach misses a known malware. False positive is produced if a benign app is identified malicious. CONFIDA results are evaluated against the known malware families, Google Play and third-party market apps.

As illustrated in Table 5.5, the overall accuracy of proposed approach is $> 95\%$. However, evaluating YZHC Trojan and Spyware families incurs 17% and 08% false

Family Type	Sensitive Feature(s)	Total Apps	True Positive Rate	False Negative,Positive Rate
Dendroid	SMS, Call, Audio, Video, Picture	30	100%	00, –
DogWars	SMS	37	100%	00, –
FakePlayer	SMS	38	91%	09, –
GamblerSMS	SMS	39	100%	00, –
GGTracker	SMS	35	100%	00, –
GPSSMSSpy	SMS	26	100%	00, –
HippoSMS	SMS	34	100%	00, –
Pjapps	SMS, Audio, Picture	45	100%	00,–
RogueLemon	SMS	31	100%	00, –
RogueSPPush	SMS	33	100%	00, –
SMS Boxer	SMS	288	94%	06,–
SMS Foncy	SMS	25	89%	11, –
SMS Replicator	SMS	11	100%	00, –
SMS Trojans & Spyware	SMS, Call, Audio	257	92%	08, 00
YZHC	SMS, Call, Audio	22	83%	17,–
Benign Apps	SMS, Call, Audio, Video, Picture	1157	97.9%	–, 2.1%
Total		**2108**	**97.7%**	**2.3, 2.1%**

Table 5.5: Evaluating CONFIDA. (FN, – for malware evaluation); (–, FP for benign app evaluation).

alarm respectively. The manual analysis reveals reveals the reason. The use of dynamic code loading to evade static analysis is the cause of false negatives. In addition, reflection API not resolved statically contribute to the false negative. The aggregate detection rate is still above > 95% suggesting CONFIDA can be deployed as a static app vetting tool.

5.3.3 Evaluating Google Play apps

Here, we illustrate two Google Play apps invoking sensitive functionality with and without user interaction. Figure 5.4 illustrates the app `cn.menue.superredial` with a single feature misuse path. The app has declared pre-specified number redial facility. The app also has a covert picture click feature misuse path not specified in the documentation. The second app `com.wn.message` is a typical SMS app. We observe that all paths get initiated from user triggered events.

As depicted in Table 5.5, CONFIDA experiment evaluated 537 Google Play apps, out of which 11 suspicious samples were reported. To validate the same, we sub-

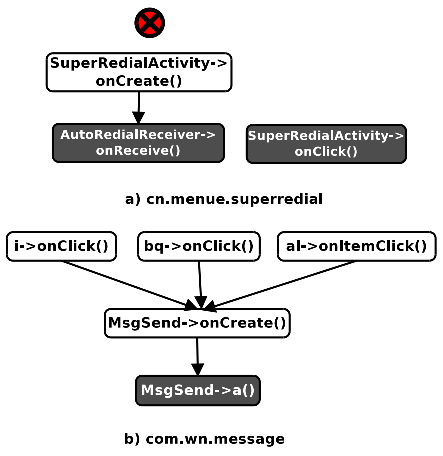


Figure 5.4: Feature misuse in benign Google Play app.

mitted the apps at *VirusTotal* [148]. The commercial anti-malware reported 7 malicious apps. The remaining were detected benign. The false positive generated by CONFIDA approach `records audio, dials a prespecified number` without user consent. These apps have declared the said functionality at install time and already known to the user indicating CONFIDA has low false alarm against unseen malware. For example, `cn.menue.superredial` app permits only a single phone number for automatic redial. Similarly, banking apps and spying apps may also have auto-redial and auto-send SMS functionality. CONFIDA detects such actions and labels them malicious.

5.3.4 Evaluating Obfuscated apps

To substantiate that CONFIDA is resilient against trivial code obfuscation, we compared the proposed approach with commercial anti-malware on the code obfuscated malware. The obfuscated variants were produced using popular `x86` transforms like string encryption, method renaming, register renaming [14]. Table 5.6 illustrates the experimental evaluation of the proposed approach against the top anti-malware products. We chose three known malware instance from each family, transformed them to evaluate CONFIDA.

As illustrated in Table 5.6, commercial anti-malware are easily evaded by the obfuscated variants of known malware. The commercial anti-malware have a poor detection rate ranging between 7-35%. Compared to the existing commercial anti-malware and analysis techniques, CONFIDA is resilient to simple code transformation techniques.Tick mark ✔ indicates that the compared methods can detect the transformed samples. The cross ✗ means that the compared tools is evaded by the obfuscated malware. Table 5.6 exhibit the CONFIDA resilience with trivial code obfuscation. Hence, the proposed approach can be used as an automated app vetting tool.

Malware Family	**Avast**	**AVG**	**ESET**	**Dr. Web**	**Kaspersky**	**McAfee**	**Symantec**	**Proposed CONFIDA**
Dendroid	✔	✗	✗	✗	✗	✗	✗	✔
DogWars	✗	✗	✔	✗	✗	✗	✔	✔
FakePlayer	✗	✗	✗	✗	✗	✔	✗	✔
GamblerSMS	✗	✔	✗	✗	✗	✗	✗	✔
GGTracker	✗	✗	✗	✗	✗	✗	✗	✔
GPSSMSSpy	✔	✗	✗	✗	✗	✔	✗	✔
HippoSMS	✗	✔	✗	✗	✗	✗	✔	✔
PJapps	✗	✗	✗	✗	✗	✗	✗	✔
RogueLemon	✗	✗	✗	✗	✗	✔	✗	✔
RogueSPPush	✔	✔	✔	✗	✗	✗	✗	✔
SMSBoxer	✔	✗	✗	✗	✗	✔	✔	✔
SMSFoncy	✗	✗	✗	✗	✗	✗	✗	✔
SMSReplicator	✗	✗	✗	✗	✔	✗	✗	✔
jSMSHider	✗	✗	✗	✗	✗	✔	✗	✔
SMS Trojan and Spyware	✗	✔	✗	✔	✔	✗	✔	✔
Detection %	21.4	28.57	14.28	7.14	14.28	35.71	28.57	100

Table 5.6: Evaluating CONFIDA and anti-malware with obfuscated malware.

5.4 Discussions

CONFIDA performs rich semantic information compared to the existing state of the art. CONFIDA extracts a combination of synchronous, asynchronous and ICC API to generate precise information compared to the current program artifacts. We detect malicious applications performing covert behavior from Google Play so far evading the existing analysis techniques. Moreover, the proposed approach identifies feature misuse among the Google Play and third-party app markets. The proposed novel analysis technique identifies a new class of evolving malware implementing covert behaviors. Such samples remain undetected with the existing state of the art and commercial anti-malware.

However, CONFIDA is not a panacea for the mobile malware detection and has its own share of limitations. The proposed analysis technique checks the covert behaviors by relating the sensitive feature misuse with corresponding user triggered inputs. If CONFIDA finds the correct relation, it concludes that the feature use is legitimate. If a Trojan apps sends SMS messages tricks the user to click with social engineering technique, the proposed method considers such action as user consented and hence legitimate. For example, a more recent variant of `FakeInst` malware tricks users to purchase paid content by clicking on a UI element (e.g., a button). However, the information provided is fake. The UI element sends an SMS to a premium-rate number without user knowledge. The cases where a user is tricked to click are considered out of scope of the proposed CONFIDA.

5.4.1 Comparison with state-of-the-art

Table 5.7 illustrates a comparison of proposed CONFIDA with similar approaches in the literature. There are quite a few static and dynamic analysis approaches for analysis, security assessment, data-flow analysis or app consistency. The IccTA leverages static taint analysis to identify tainted variables (sources) and trace them to the possible vulnerable functions (sinks) to detect sensitive data ex-filtration. However, we propose an ICC based control-flow analysis on a reasonable dataset. The proposed approach analysis performs better compared to the existing methods reported in literature, as illustrated in Table 5.7.

Detector	Target	Analysis Technique	Features	Classification Technique	# Samples	Repository	Accuracy
AndroLeaks [149]	Confidentiality	Static	Dangerous API	Map sensitive data with dangerous API	B#: 24000	Google Play, Third party markets	FP: 35%
Andromaly [58]	Anomaly detection	Dynamic	Device features	Tree based machine learning classification	M#: 4	Self developed malware	FP:17.8%
Amos et al. [150]	Malware classification	Dynamic	CPU, memory, battery features	Real-time testing	Train: M#:1400,B#:49 Test: M#:24 B#: 23	GNOME, VirusTotal, Google Play	FP: 15% FN: 12%
AndroSimilar [101]	Signature based variant detector	Static	Byte based statistical similarity	Robust features for malware variants	B#: 3300 M#: 2200	GNOME,VirusShare, Google Play	FP: 5% FN: 6%
AsDroid [95]	UI based stealthy behavior	Static	UI text mapping with stealthy behavior	WALA based java code analysis	B#: 74 M#: 96	GNOME,VirusShare, Google Play	TP: 85% DR: 88%
CrowDroid [93]	Malware classification	Dynamic	System call traces	Crowdsourcing based system call trace	developed#: 03 real M#: 02	VirusTotal, Synthetic malware	FP: 20%
Drebin [151]	Malware classification	Static	Permissions,Hardware, Network, URL, API	multiple APK features	B#: 0.12 million M#: 5560	Google Play,GNOME, Malware blogs	FN: 6% FP: 1%
IccTA [84]	Sensitive data ex-filtration	Static	Inter-component privacy leakage	ICC Taint Analysis	B#: 15,000 M#: 1260	Google Play,GNOME	PR#: 96.6% RC#: 96.6%
Elish et al. [96]	Anomaly of User initiated triggers	Static	ICC Control and Data-flow	user initiated triggers	B#: 2684 M#: 1433	Google Play,GNOME, VirusShare	FP: 2.0% FN: 2.1%
RiskRanker [28]	Anomalous code and behavior	Signature, Heuristics	Behavior tracing	weight based features	B#: 0.118 million	GooglePlay, Third party markets	FN: 9%
FlowDroid [86]	Sensitive data ex-filtration	Static	Sensitive user data	Field and object based taint analysis	V#: 150	DroidBench	RC: 93%
Proposed CONFIDA	*Covert behavior with sensitive API*	*Static*	*ICC based control and data flow*	*Anomalous covert, malicious behavior*	*B#: 1157 M#: 951*	*GooglePlay,GNOME, VirusShare*	*FP:2.1%, FN: 2.3%*

Table 5.7: CONFIDA comparison with recent analysis techniques.

Table 5.7 compares the analysis techniques based on the following parameters. The detection technique targets a particular malware app class, analysis method employed, features extracted, number of samples and reported accuracy. The

AndroLeaks targets the confidential user data by mapping the dangerous API calls based on requested permissions on 2,400 benign apps. The analysis technique incurs 35% false positive. Andromaly is an on-device dynamic analysis framework to detect malicious apps based on important features. The authors evaluated 7 self developed synthetic malware reporting 17.8% FP. Our proposal considers control and data flow analysis to identify invocation of sensitive feature misuse incurring a low false alarm compared to the existing work in literature.

5.5 Summary

The Android OS is a prominent platform for the emerging technologies like Internet of Things (IoT) and smart city infrastructure services. The malware authors are targeting the Android platform with evasive techniques, providing fertile ground for malware attacks. In this chapter, we proposed *CONFIDA*, a high precision inter-component communication based detection of sensitive feature misuse perpetrated by the evolving Android malware. The proposed approach identifies sensitive functionality necessitating explicit user intervention. We generate a precise Dalvik bytecode analysis technique considering the component-interaction graph (CIG) and Android ICC API.

Furthermore, we perform reverse reachability analysis to identify if the *feature usage* or *behavior* is initiated by the legitimate user interaction. The encouraging results on the dataset of about one thousand notable SMS and spyware indicate that CONFIDA can be deployed as a static app vetting framework for Android app market. We evaluated 951 malicious and 1,157 benign apps with classification accuracy with 2.3% false negative rate and 2.1% False Positive Rate (FPR), superior than the existing approaches.

Chapter 6

Evaluating Anti-malware against Dalvik bytecode Obfuscation

In the previous chapter, we performed static evaluation of the covert behavior employed by the Android malware. However, if the malware author employs code obfuscation, static analysis methods must perform the de-obfuscation before analysis. In this Chapter, we evaluate the capabilities of code obfuscation on Android apps to identify the resilience of existing anti-malware and static analysis techniques. Furthermore, we evaluate our existing proposal AndroSimilar, a robust statistical feature signature against Dalvik bytecode obfuscation.

6.1 Dalvik Bytecode Obfuscation

Code obfuscation has been reported as an alternative code protection technique. The Code obfuscation is intended to render the code unreadable or at least make the original code difficult to decipher. Code obfuscation transforms the code by changing its physical appearance while preserving the intended program behavior. In short, code obfuscation can be used to protect the software from reverse engineering. The malware developers have also leveraged the obfuscation in developing recent malware apps [113, 152] evading the commercial anti-malware. The rich bytecode semantics and easy availability of reverse engineering tools contribute to the exponential increase of Android malware [153]. Obfuscated malware threat has prompted the requirement of robust anti-malware techniques.

Code obfuscation is treated as changing control flow, data flow or layout changes of an executable maintaining the original semantics. Malware writers use code transformation methods to propagate unseen variants and evade the anti-malware. Malware app, M_{app} consists of random malicious functions like sending SMS without user consent, ex-filtrating sensitive user data without his knowledge. The malicious app is defined as:

$$M_{app} = m_1, m_2, m_3, \cdots, m_n. \tag{6.1}$$

Here, $\mathrm{m_1}$, $\mathrm{m_2}$ .. $\mathrm{m_n}$ are functions within an app where one or more functions are malicious. A variant may introduce arbitrary number of additions, modifications or deletions in the code by inserting malicious functions, maintaining the original semantics. A malware variant is represented as:

$$M_{app} = m_1, m_{mod\_2}, m_3, m_{add\_1}, m_{add\_2}, \cdots, m_n, \tag{6.2}$$

where m_{add_1}, m_{add_2} are the new functions inserted in the code and m_{mod_2} is the corresponding code modification m_2, maintaining the intended functionality. The newly added function may not necessarily be malicious. In general, anti-malware solutions use the blacklisting approach to tackle malware and avoid false positives (i.e., detection of an innocent app as malware). The malware authors anticipate this fact to bypass such lists and generate obfuscated variants of the existing malware.

The Dalvik bytecode executes inside the DVM. The complete type information availability makes Dalvik bytecode amenable to reverse-engineering and code obfuscation attacks. *apktool* is used to disassemble and convert Dalvik bytecode into intermediate smali mnemonics [42]. After making changes, the same tool can be used for assembling the obfuscated app. Dalvik bytecode can be retargeted to Java bytecode [154] subject to the availability of intermediate code obfuscator [48].

Rastogi et al. [155] proposed DroidChameleon, a trivial code obfuscator on a small set of 5 malicious apps against top commercial anti-malware. The authors employ multiple obfuscation in sequence to evade the anti-malware. The authors randomly selected trivial transformation techniques without considering the representative Control, Data and Layout obfuscation class. Our proposal identifies representative obfuscation techniques from the 3 categories and evaluates static analysis techniques and commercial anti-malware performance. Furthermore, we select a reasonable dataset for evaluation compared to the existing analysis techniques DroidChameleon [155] and ADAM [47] reported in the literature.

6.2 DroidsHornet: Dalvik Bytecode Obfuscator

In this section, we discuss the implementation of the obfuscation prototype and evaluate our proposed static analysis technique against commercial anti-malware and static analysis techniques. The existing analysis techniques retarget the Dalvik bytecode to Java constructs for analysis. The conversion process eliminates important fine-grained details present in the intermediate bytecode. To preserve the finer details, our proposal obfuscates the bytecode.

Collberg et al. [156] classify the obfuscation techniques as: (1) Control flow; (2)Data flow; and (3) Layout. Control-flow (CT) obfuscation aims to confuse the analyst by changing the control-flow of the source code. The functional blocks are broken apart to confuse the reverse engineering. The Data obfuscation (DT) techniques modifies the structure by modifying the data values. The Layout transformation (LT) targets the lexical structure of the program such as variable names or formatting the source code.

6.2.1 Proposed Obfuscator Design

Figure 6.1 illustrates the automated procedure for app disassembly, code transformation and APK re-assembling. The obfuscated apps are evaluated against AndroSimilar [101] our proposal for obfuscated app detection and Androguard [52] a state-of-the-art static analysis tool. The proposed Transformation evaluates the performance of existing techniques against Control (CT), Data (DT) and Layout (LT) transformations. We evaluate the performance of anti-malware solutions on apps obfuscated with different permutations of Control, Data and Layout techniques.

Figure 6.1 illustrates the automated procedure for generating the code obfuscated variants. An input APK is disassembled with APKTool [42] to convert the bytecode in smali format. The smali file represents intermediate code of the Java class declared in the app. In the next step, our code obfuscation program modifies the

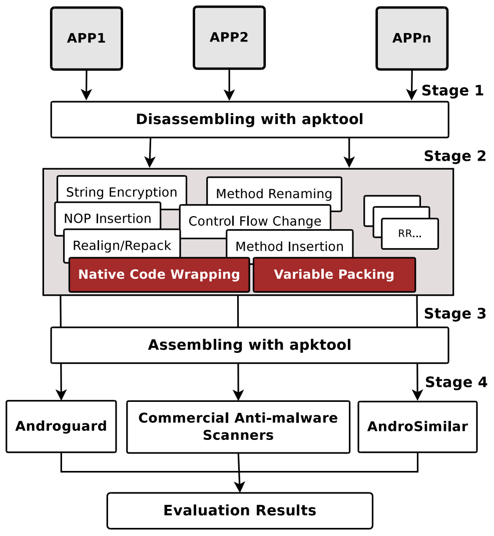


Figure 6.1: Proposed analysis Technique.

smali code with the implemented obfuscation techniques. The modified smali code is assembled and re-signed. The new APK file is the semantic equivalent of the original malware with a different syntax structure. The disassembling procedure, code obfuscation and APK regeneration is automated in the prototype. The original and modified APK samples are submitted at *VirusTotal* [148] to evaluate the commercial anti-malware.

In the following, we discuss the implemented code obfuscation techniques and evaluate the obfuscation effect the analysis technique.

6.3 Control Transformations

The Control flow obfuscation breaks up the control flow of the source code. Functional blocks broken apart or intermingled to confuse the reverse engineering. Control flow transformation changes the execution paths of a program, maintaining its intended functionality.

6.3.1 Altering Control Flow

Listing 6.2 illustrates control flow obfuscation by inserting unconditional jumps to circumvent the disassembly and evade static analysis.

```
//GoldDream malware unobfuscated code
.method private IsClearLocalWatchFiles()V
 .locals 2
 .annotation system Ldalvik/annonation/Throws;
 value={ Ljava/io/IOException;}
 .end annotation
...
 .line 224
 .local v1,"objSmsFile;Ljava/lang/String;"
 const-string v0,"/data/data/com.dchoc.tuxdo/files/zjphonecall.txt"
...
.line 227
.local v0,"objCallFile;Ljava/lang/String;"
 invoke-direct {p0,v1},Lcom/GoldDream/zj/zjService;>CheckAndClearFile(Ljava/lang/String;)V
...
.end method
```

Listing 6.1: GoldDream malware smali code before obfuscation.

The smali code of a disassembled Java class is illustrated in the above listing. When we employ control flow obfuscation, the `goto` unconditional jump instruction changes the flow of a particular class.

```
//Control-flow obfuscation
.method private IsClearLocalWatchFiles()V
.locals 2
.annotation system Ldalvik/annonation/Throws;
 value={ Ljava/io/IOException;}
.end annotation
.prologue

  goto :goto_1
  .line 223
  goto :goto_0
  goto :goto_0
  const-string v1,"/data/data/com.dchoc.tuxdo/files.zjsms.txt"
...
 goto_1
 goto : goto_0
...
.end method
```

Listing 6.2: After control flow obfuscation.

The Control flow represents the path an app may traverse during execution. We alter the control flow of the bytecode, changing the file signature. The bytecode modifications change the structure without affecting the original functionality.

6.3.2 No Operation (NOP) Insertion

The No-operation code instruction is used to change the opcode sequence and thwart the $n - gram$ based static analysis. Insertion of `nop` instruction also wastes the CPU execution cycle. The Dalvik `nop` instruction is randomly added to the disassembled methods preserving the semantics. This obfuscation evades anti-malware solutions employing Dalvik opcode sequence as malware signature. Listing 6.1 and 6.3 illustrates the original code and `nop` insertion respectively.

```
//no operation code $nop$ insertion
.method private IsClearLocalWatchFiles()V
      nop
      nop
      .locals 2
 ...
      nop
     .prologue
 ...
 .line 224
 .local v1,"objSmsFile;Ljava/lang/String;"
      nop
...
.end method
```

Listing 6.3: No Operation obfuscation.

6.3.3 Dead Code Injection

Inserting dead-code is another technique to evade the analysis. A programmer can insert code that is never executed or may not contribute to the functionality of the program [157]. This code can include extra methods or few lines of irrelevant code [158]. For instance, the code snippet before the Add Dead-code Switch Statements (ADSS) [101] is illustrated in the Listing 6.4.

The Java bytecode switch construct can be used to insert control flow switch that is never executed [159]. However, the switch increases the connectedness and complexity of the method. Thus, the obfuscation evades the decompiler that cannot remove the dead switch. Listing 6.5 illustrates the ADSS obfuscation [159].

```java
//before inserting ADSS obfuscation
if (writeImage != null) {
try {
  File file = new File("out");
  ImageIO.write(writeImage, "png", file);
 }
 catch (Exception e) {
 System.exit(1);
 }
}
 System.exit(0);
```

Listing 6.4: before ADSS.

```java
// ADSS obfuscated code

if(obj != null) {
 try {

  ImageIO.write((RenderedImage)obj,png,
  new File(out));
    }

  catch(Exception exception2)  {
   ++i;
   obj = exception2;
   i += 2;
   System.exit(1);
    }
}

label_167:

{ while(1I1.booleanValue() == ___)
  {
    switch (i) {
    default: break;
    case 3: break label_167;
    case 1: ++i;
    obj = exception2;
    i += 2;
    System.exit(1);
    continue;
    case 2:
    i += 2;
    System.exit(1);
    continue;
       }
    }
    System.exit(0);
}
```

Listing 6.5: ADSS Obfuscation [160].

6.4 Data Transformation

6.4.1 Realign/Repack

zipalign tool, a part of Android SDK realigns the app for better performance. In case of repacking, an app is disassembled using *apktool*, reassembled and re-signed without any bytecode changes. Such trivial changes evades the commercial anti-malware signature. Listing 6.6 illustrates *GoldDream* malware with realigned obfuscation. It is important to note that realigning the APK has no effect on the smali code.

```
// APK realign
.method private IsClearLocalWatchFiles()V
 .locals 2
 .annotation system Ldalvik/annonation/Throws;
 value={ Ljava/io/IOException;}
 .end annotation
...
 .line 224
 .local v1,"objSmsFile;Ljava/lang/String;"
 const-string v0,"/data/data/com.dchoc.tuxdo/files/zjphonecall.txt"
...
.line 227
.local v0,"objCallFile;Ljava/lang/String;"
 invoke-direct {p0,v1},Lcom/GoldDream/zj/zjService;>CheckAndClearFile(Ljava/lang/String;)V
...
.end method
```

Listing 6.6: No effect of Realign obfuscation.

6.4.2 Variable Compression

`x86` platform executable compressors are used to pack malware payload inside an arbitrary section of the target file. Malware payload is uncompressed when the file is loaded into the memory. We implement data flow transformation by encasing the numeric constants. This transformation is based on opaque condition [156] where numeric constants are compressed into an external packer class.

6.4.3 Native Call Wrapping

Native libraries (.so) are used to perform CPU intensive tasks. For a unique method, a wrapper function is constructed to redirect native calls. Call wrapping evades control flow as functions call to the libraries are scattered. As the number of malware app using native calls low, few transformed apps are generated.

6.4.4 Resource Encryption

Content of *resources*, *assets* and native libraries can be encrypted, but needs modification within bytecode to decrypt at runtime. Android runtime Exploits [34, 35, 36] are native code based encrypted payloads to exploit the device and evade anti-malware.

6.5 Layout Transformation

6.5.1 Method Insertion

In this transformation, a dummy method is inserted in every class of Dalvik bytecode maintaining the actual behavior. Method insertion increases bytecode size and modifies the Dalvik method table to alter its binary footprint. Listing 6.7 illustrates the obfuscated bytecode after inserting dummy method in the GoldDream malware class.

```
.method public static DummyMethod(Ljava/lang/String;Ljava/lang/String;)V
/lang/String;Ljava/lang/String;)V
 .register 2
 .parameter ‘‘tag’’
 .parameter ‘‘msg’’
 .line 28
invoke-static p0,p1.Landroid/util/Log;   > d(Ljava/lang/String;Ljava/lang/String;)I
 .line 29
return-void
.end method
. method private IsClearLocalWatchFiles() V
 .....
. prologue
 ...
.end method
```

Listing 6.7: Method Insertion.

6.5.2 String Encryption

String encryption encodes literal strings and renders them unreadable. This transformation decrypts encoded strings during the app execution. We store the strings in a byte array, encrypt and store the bytes instead of strings. Transformed strings are not stored inside the *string_ids*, making it hard to discover. Listings 6.1 illustrates original malware code and Listing 6.8 illustrate effect of string encryption.

```
 const-string v1,"nkdk/nkdk/myw.nmrym/dehny/psvoc/jtcwc.dhd"
  invoke-static(v1), Lcom/mnit/jaipur;$->$
  Decrypt(Ljava/lang/String;) Ljava/lang/String;}}*)
  move-result-object v1
  ...
  invoke-static v0, Lcom/mnit/jaipur;$->$
  Decrypt(Ljava/lang/String;) Ljava/lang/String;
  move-result-object v0
.line 227
.local v0,"objCallFile;Ljava/lang/String;"
 invoke-direct (p0,v1), Lcom/GoldDream/zj/zjService;>CheckAndClearFile(Ljava/lang/String;)V
...
 .line 229
   .return-void
 .end method
```

Listing 6.8: String Encryption.

6.6 AndroSimilar Signature

AndroSimilar signature generation approach is based on the hypothesis that two unrelated files have a low probability of having common features. Fixed-size byte sequence features are extracted based on the empirical probability of occurrence of their entropy values. The values are computed in a sliding window fashion. The popular features are identified according to their neighborhood rarity [137]. When two unrelated files share some characteristics, the features are considered weak contributing to the false positives [138]. Initially, we generate signatures of known malware families in the existing representative malware database. Then, we compare the similarity score of an unknown app with the existing database. If the signature database matches the unknown signature beyond an experimental threshold, the application is labeled malicious. In the following, we evaluate and compare our proposed approach with commercial anti-malware and Androguard, a robust static analysis technique.

6.7 Experimental Evaluation

In this Section, we evaluate the obfuscated apps against our proposal AndroSimilar, Androguard static analysis tool and commercial anti-malware. We obfuscate the benign Google Play apps and malware samples from known families discussed in Section 6.7.1. Figure 6.2 gives an overview of the proposed transformation to generate new malware variants automatically. An app is disassembled with

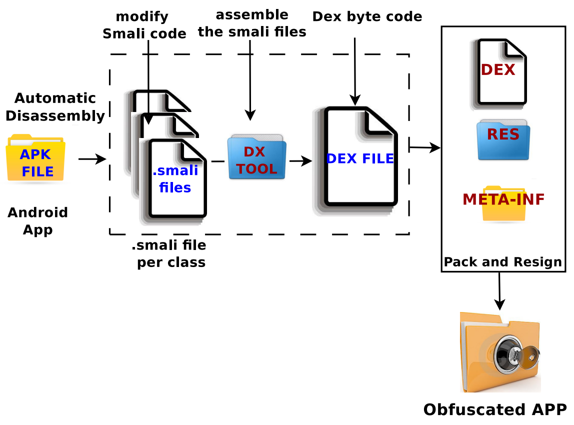


Figure 6.2: Evaluating Transformation techniques.

Baksmali [161] where each class is represented by a `.smali` file preceded with a $. Smali stores extracted Dalvik bytecode mnemonics. We transform smali mnemonics and assemble all the classes into a single Dalvik EXecutable (DEX). DEX, resources and meta information is re-signed to generate a obfuscated app. Authors in [47, 155] randomly select trivial techniques and bytecode transforms whereas proposed evaluation prototype considers representative samples of Control, Data and Layout transformation techniques discussed in [162].

6.7.1 Dataset for Evaluation

Table 6.1 illustrates the data source whereas, Table 6.2 lists benign apps and known malware families. We select 15 malware families from Genome Project [2] with 10 representatives samples from each family. Samples were downloaded from VirusShare [128] and Contagiominidump [127] to add representative malware confirming their signature with commercial anti-malware engines. We have generated 764 malware variants using the proposed obfuscation prototype.

Source	**# of Samples**
Malware Genome Project	150
VirusShare	10
Contagiominidump	10

Table 6.1: Android Malware Dataset.

Obfuscated apps are evaluated against 52 commercial anti-malware on Virustotal [148]. We compare the performance of obfuscated apps with Androguard code similarity module. The method calculates Normalized Compression Distance (NCD). The AndroSimilar is based on SDHash algorithm to identify obfuscated malware using robust statistical signature. The evaluation is carried out against code obfuscated malware apps and Trojanized Google Play apps. Transformed apps are evaluated with 52 commercial anti-malware at *VirusTotal*, a web based anti-malware interface.

6.7.2 Google Play apps

According to [157], profit motives and poor code protection techniques expose the apps for misuse and abuse from malware authors. We selected two popular apps with minimum 10,000 downloads from 32 Google Play categories. The selected apps were obfuscated and evaluated against static analysis techniques.

Malware Source & Families		**Google Play Apps**		
AnserverBot	GoldDream	Arcade & Action	Entertainment	Personalization
BeanBot	HippoSMS	Books & Reference	Finance	Photography
DroidKungFu	Kmin	Brain & Puzzle	Health & Fitness	Productivity
DroidKunguFuUpdate	PJApps	Business	Libraries & Demo	Racing
DroidKungFu2	SpitMO	Cards & Casino	Lifestyle	Shopping
EndofDay	ZitMO	Casual	Media & Video	Tools
FakeNetFlix	**Contagiominidump**	Comics	Medical	Transportation
FakePlayer	**VirusShare**	Communication	Music & Audio	Travel & Local
GGTracker		Education	News & Magazines	✗ weather
# Apps per Family	**10**	**# Apps per Category**		**2**

Table 6.2: Google Play & Malware dataset.

6.7.3 Evaluating anti-malware techniques

Evaluation results of anti-malware are shown in Table 6.3. Original and obfuscated apps were submitted to Virustotal [148] in first two weeks of March 2014. Table 6.3 illustrates aggregated 52 anti-malware evaluation. Table 6.3 second column evaluates 52 commercial anti-malware detection. The unobfuscated Anserver malware samples detection rate is 88%. The results indicates commercial anti-malware do not perform frequent signature updates.

Family Name	**Original**	**Repacking**	**Obfuscations Methods**						
			DC	**CF**	**MI**	**SE**	**VP**	**NCW**	**NOP**
AnserverBot	88	42	30	34	38	36	23	27	31
BeanBot	72	38	22	28	36	32	32	29	26
DroidKungFu	76	46	54	28	38	34	31	30	27
DroidKunguFuUpdate	52	28	22	20	32	32	33	25	23
DroidKungFu2	82	44	24	22	38	34	30	28	24
EndofDay	86	48	20	22	40	36	24	26	24
FakeNetFlix	76	38	22	20	36	38	27	27	24
FakePlayer	82	40	26	24	38	38	31	32	29
GGTracker	74	48	34	24	36	34	26	24	25
GoldDream	68	40	32	22	32	38	23	21	22
HippoSMS	74	40	28	32	30	38	24	23	26
Kmin	60	48	36	30	44	42	22	26	28
PJApps	74	44	20	24	42	36	23	23	24
SpitMO	78	46	32	30	30	36	34	31	32
ZitMO	78	42	32	32	36	32	31	29	26
Contagiominidump	72	38	20	26	32	42	30	23	26
VirusShare	54	40	28	32	34	38	26	23	21

Table 6.3: Cumulative detection (%). DC–Dead Code, CF–Control-flow altering, NOP–No operation code, MI–Method Insertion, SE–String Encryption, NCW–Native code Wrapping, RR–Register reassignment VP–Variable compression.

When we re-evaluated the repacked apps, the anti-malware detection drops by 50%. Testing code obfuscated malware families fails *maximum number of anti-malware* with aggregate detection dropping to 21%. Top rated anti-malware *Avast*, *AVG* and *Dr. Web* performed comparatively better. The impact of code transformation against additional methods is illustrated with background color. Evaluation against AnserverBot family malware drops to 30% in case of simple *nop* obfuscation.

The trivial obfuscation techniques variable compression and *nop* evades the commercial Android anti-malware. Similar trends are visible for Control Flow altering, Method Insertion and String Encryption. The samples evaluated with static analysis techniques AndroSimilar and Androguard outperform the commercial anti-malware. AndroSimilar, our proposed malware variant detection approach performs better than commercial anti-malware. When the obfuscated samples are evaluated against Androguard, it detects the transformed samples with reasonable accuracy.

6.7.4 Commercial anti-malware & Analysis Techniques

Moving further, we evaluate and compare top commercial anti-malware [163] against static analysis techniques rather discussing aggregated results. This procedure tests the resilience of anti-malware against additional techniques implemented so far on `x86` platform. Background color illustrates performance of *AntiY*, *ClamAV*, *Microsoft* and *Symantec* as poor. Testing *Avast*, *AVG* and *Dr. Web* results show that they are resilient against individual transformation techniques.

Additional Methods	Anti Malware Products (AM)								Static Analysis Tools		
	AntiY	Avast	AVG	ClamAV	Dr. Web	McAFee	Microsoft	Symantec	Trend Micro	Andro guard	Andro Similar
NOP	✗	✔	✔	✗	✔	✗	✗	✗	✗	✔	✔
NCW	✗	✔	✔	✗	✔	✔	✔	✔	✔	✔	✔
RR	✗	✔	✔	✔	✗	✔	✔	✗	✗	✔	✔
VP	✗	✗	✗	✗	✗	✗	✗	✗	✗	✔	✔
NOP+NCW	✗	✔	✔	✗	✔	✗	✗	✗	✗	✔	✔
NCW+RR	✗	✗	✔	✗	✔	✗	✗	✗	✗	✔	✔
NCW+VP	✗	✔	✔	✗	✔	✗	✗	✗	✗	✔	✔
NOP+VP +RR	✗	✗	✔	✗	✗	✗	✗	✗	✗	✗	✗
NOP+NCW +RR+VP	✗	✗	✗	✗	✗	✗	✗	✗	✗	✔	✗

Table 6.4: Evaluation:✗: anti-malware evaded by obfuscation. ✔ Technique is resilient. NOP–No operation code, NCW–Native code Wrapping, RR–Register re-assignment VP–Variable packing/compression.

Table 6.4 compares the performance of AndroSimilar with popular commercial anti-malware employing CT, DT and LT transformations and their combinations thereof. In particular, *Avast* fails to detect variable compression and combination of control, data, and layout obfuscation. We employ a combination of Control, Data and Layout transformation rather than evaluating repetitive transformations. *AVG* and *Dr. Web* exhibit similar trends with poor detection rate against the additional obfuscation methods.

Table 6.4 illustrates the comparative performance of AndroSimilar, our proposed solution and static analysis tool, Androguard. Both analysis techniques outperform the commercial anti-malware. Encrypted code and dynamic loading obfuscation evade the Androguard. The only case where Androguard fails is a combination of *nop, VP and RR*. AndroSimilar, a robust file based statistical signature based approach outperforms the commercial anti-malware.

Here, we discuss the performance comparison of Control, Data and Layout transformation methods against anti-malware techniques. Table 6.5 depicts the evaluation of techniques individually, and permutations of C, D and L methods. The known anti-malware *AntiY, ClamAV, McAfee, Symantec, and Trend micro* are evaded by the combinations of control, data, and layout transforms. Depicted in row 2 and 3, *Avast* and *AVG* fail against permutation of control, data, and layout transformation. Dr. Web is reasonably resistant to code transformation, suggesting that top rated anti-malware incorporate robust feature selection methods.

Anti Malware	**Transformation Class**			**Combined Classes**			
	C	D	L	C + D	C + L	D + L	C + D + L
AntiY	✗	✗	✗	✗	✗	✗	✗
Avast						✗	✗
AVG							✗
Clam-AV	✗	✗	✗	✗	✗	✗	✗
Dr. Web				✗		✗	✗
McAFee	✗			✗	✗	✗	✗
Microsoft	✗	✗	✗	✗	✗	✗	✗
Symantec	✗	✗	✗	✗	✗	✗	✗
TrendMicro	✗	✗	✗	✗	✗	✗	✗
Androguard							✗
AndroSimilar			✗				✗

Table 6.5: Evaluating anti-malware Techniques. ✗: transformation evaded. C–Control, D–Data and L–Layout Transforms.

Androguard and AndroSimilar outperform the top commercial anti-malware consistently against permutations of control, data, and layout categories. Table 6.5 illustrates that static analysis techniques can evade with a combination of multiple obfuscation techniques. AndroSimilar generates robust signatures by extracting statistically robust features to detect malicious apps.

AndroSimilar finds regions of statistical similarity with known malware to detect unknown malware variants using syntactic similarity, rather than embedded DEX file employed by known fuzzy hashing approaches. Androguard is a semantics normalized compression distance based software similarity tool to identify the similarity among malware variants. As the approach is NCD based, it takes more time to identify similar methods compared to AndroSimilar.

Figure 6.3 illustrates the resilience of AndroSimilar and Androguard. The cumulative data gathered suggest that *AntiY* performs worst among all commercial anti-malware. *Avast* and *AVG* perform comparatively better. Androguard and AndroSimilar outperform the commercial anti-malware. This evaluation suggests improvements and remedial solutions to perform effective malware detection.

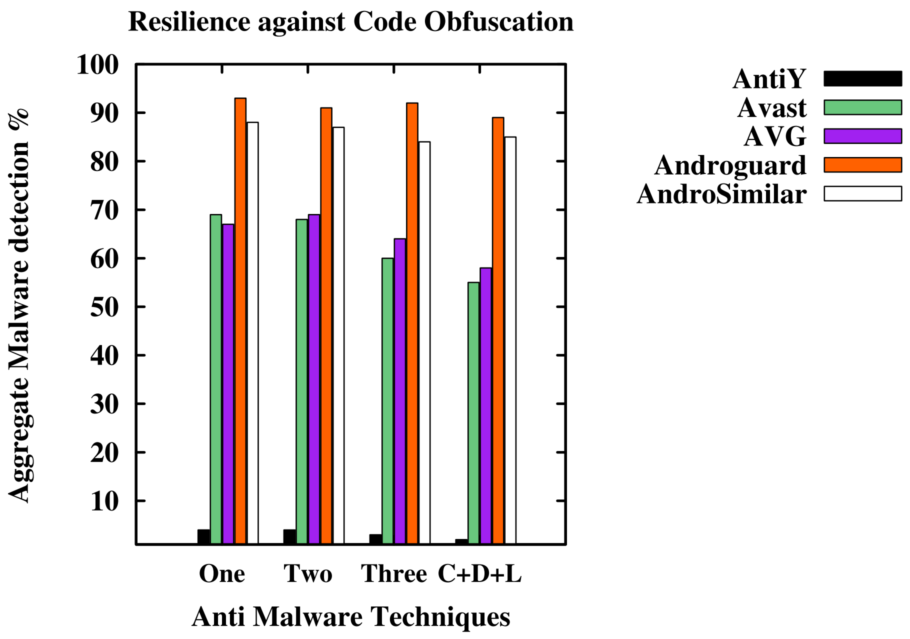


Figure 6.3: Resilience of anti-malware and static analysis Techniques.
C = Control, D = Data, L = Layout, Transforms.

6.8 Discussions

In this Section, we discuss the important observations during the experimental evaluation of anti-malware and our proposal AndroSimilar.

1. ADAM [47] reports commercial anti-malware *AntiY* effective at analyzing code obfuscated malware. We have experimentally evaluated the *AntiY* with 750 transformed samples. *AntiY*, a known anti-malware can be evaded with very trivial code obfuscation techniques illustrated in Table 6.4. Similarly, the top commercial anti-malware performed poor. The same unseen malware were evaluated at *VirusTotal* after 2 weeks of the initial submission. The results of top anti-malware improved drastically. This suggests anti-malware signature are updated regularly; hence, the signature update at intervals improve the detection.

2. The commercial anti-malware are evaded with multiple code obfuscation applied in sequence. DroidChameleon [155] employs repetitive naive obfuscation until the technique evades the anti-malware. This approach renders the malware unusable on account of repeated obfuscation. However, if the obfuscation is class based, the sample remains consistent.

3. Rastogi et al. [155] rated *AVG* as poor performer against the 5 obfuscated malware. We have evaluated nearly 750 malware. We observed that *AVG* performed much better even among the top 10 anti-malware. DroidChameleon evaluated five samples, a subset of our 760 malware app dataset.

4. The commercial anti-malware are improving the detection with heuristic techniques. However, only few have maintained consistent performance. We have evaluated AndroSimilar, our static analysis signature proposal on the same dataset. The popular features are identified according to their neighborhood rarity. Hence, we extract persistent local minima to pick the robust features in sliding window fashion. The file features with lowest presence get high rank. Similarly, extreme presence gives low rank. To avoid superfluous features, we consider only those minima that persist among the multiple adjacent windows. Identifying the app with such rare features generates a robust statistical signature.

5. The aggregated results of AndroSimilar and static analysis tool AndroSimilar outperforms the existing commercial anti-malware by a big margin. Androguard performs better due to it similarity based Dalvik bytecode analysis. The comparison of static analysis techniques suggest a need for improving the analysis and incorporate Dalvik bytecode semantics to counter app obfuscation.

6.9 Summary

In this chapter, we have evaluated the limitations of existing static analysis techniques against trivial code obfuscation. We implemented a Dalvik bytecode obfuscation techniques popular on the x86 platform. Furthermore, we evaluate the effectiveness of robust static analysis techniques AndroSimilar and Androguard. We compared the resilience of Androguard's code similarity [52] and AndroSimilar's statistical feature signature [101] against a combination of different code transforms. The limitations of static analysis techniques prompt the use of complementary dynamic analysis methods to improve the analysis coverage.

Chapter 7

Analysis environment-aware Malware Detection

As discussed in the previous chapter, malware authors are adopting Dalvik bytecode transformations, encryption and code protection methods. The transformed malware is equipped with analysis environment detection techniques to evade static analysis and signature-based detectors. These techniques are collectively defined as analysis environment detection techniques. In this Chapter, we propose an automated dynamic analysis framework to make a malware believe that it is being executed on the real Android device instead of the emulator, an alibi for development or an analysis system. We target the Android system features with modified static emulator properties and enrich the virtual device with user information. To explore the execution paths, we integrate user input simulation with *intent* broadcasts.

7.1 Dynamic analysis Sandbox

The evolving malicious apps have the capability to identify the emulated, virtual and analysis environment. Once the app identifies itself within the analysis system, it behaves benign without revealing malicious functionality. To counter the hidden malicious behavior, we execute the apps in an emulated environment augmented with capabilities to entrap the analysis environment-aware malware to reveal the hidden malicious behavior. In addition to modification of emulator properties, we monitor the (1) file operations; (2) app downloads; (3) native payload installation;

(4) encrypted strings; and (5) SMS sent/received, to correlate malicious activities predominantly present among malicious apps [102, 105].

Android app execution is event-driven, asynchronous with multiple entry triggers. The `main_activity` or the `launcher_activity` is considered default entry point of the app. Hence, the malware authors employ techniques to execute the malicious functionality from the launcher activity. The user interface gestures such as tap, pinch, swipe and keypress must be automatically triggered to initiate the app interaction. The proposed dynamic analysis sandbox incorporates multiple analysis methods as illustrated in Figure 7.1 to improve analysis coverage. When an app is submitted to the analysis sandbox, a modified and refreshed virtual device is launched. The Android Virtual Device (AVD) manager [164] permits emulator create, execution, load, save and restore states.

A real Android device stores contacts, SMS, Google Play market app and default apps, and customized user settings not available with the emulated device. Hence, to resemble a real device, we customize the emulator with (i) Google Play and default device apps; (ii) customize the wallpaper; (iii) add contacts and SMS; and (iv) customize the user settings. The modified device is launched with custom settings when a new APK is submitted for analysis. The sandbox starts the emulator(s) with a save-to-snapshot state with wallpaper, messages, contacts and setting custom device settings. Each time an app is submitted for analysis, clean emulator snapshot is loaded.

As illustrated in Figure 7.1, the Framework core controls the components for feature collection and facilitates the AVD loading. Dalvik Dynamic Instrumentation (DDI) hooking libraries attach the methods with DVM to monitor the strings.

7.1.1 Environment-aware malware detection

The evolving Android malware families defend themselves from the analysis environment to avoid revealing the malicious behavior [31]. We modify the `IMEI`, `IMSI` and other static properties that exclusively identify a virtual device. We modify the geo-location properties, system time, configure e-mail account, add images and audio/video files. Table 7.1 compares the static parameters identified to detect analysis environment. The analysis systems employ the default emulator. The malware authors identify the below listed attributes before revealing the malicious

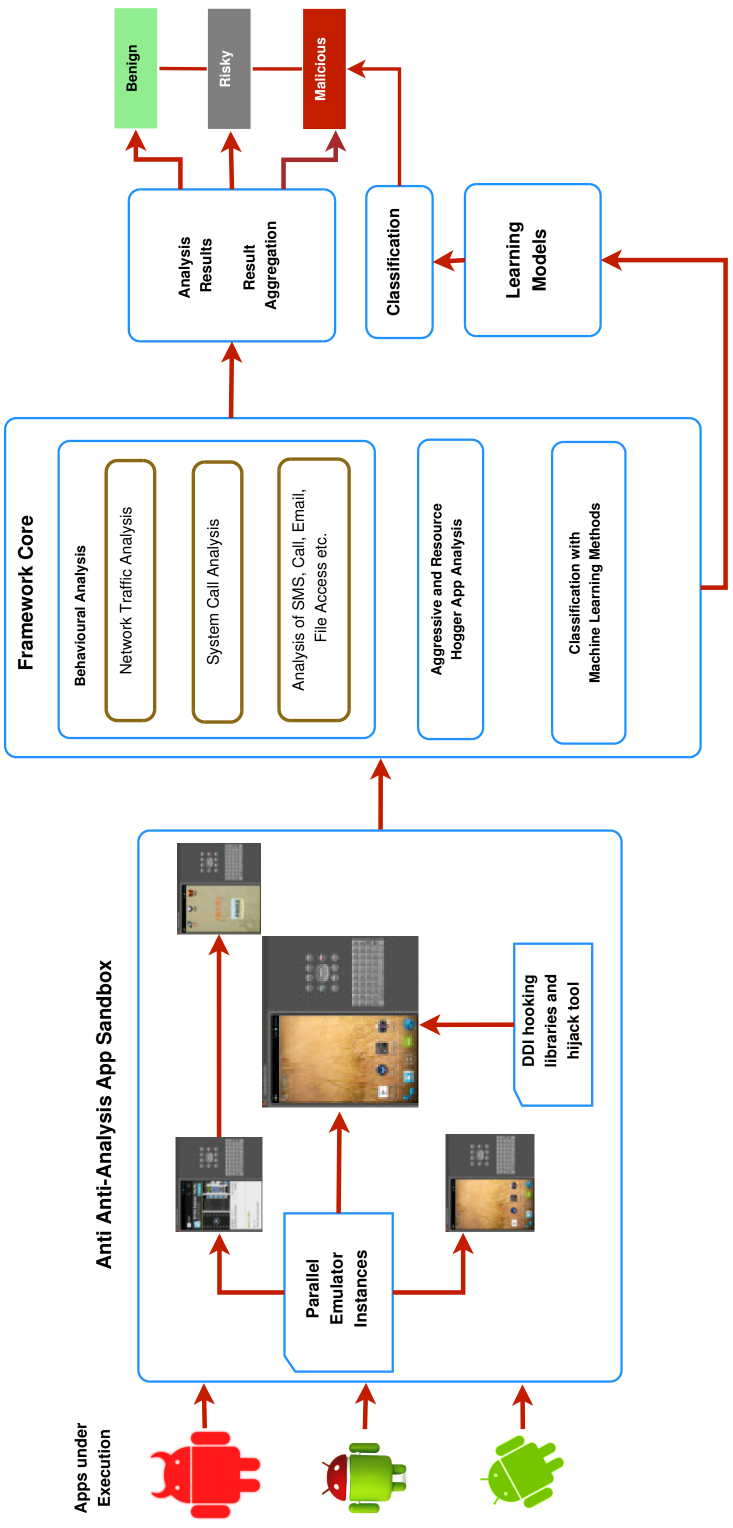

Apps under Execution
Anti Anti-Analysis App Sandbox
Parallel Emulator Instances
DDI hooking libraries and hijack tool
Framework Core
Behavioural Analysis
Network Traffic Analysis
System Call Analysis
Analysis of SMS, Call, Email, File Access etc.
Aggressive and Resource Hogger App Analysis
Classification with Machine Learning Methods
Analysis Results
Result Aggregation
Classification
Learning Models
Benign
Risky
Malicious

Figure 7.1: Proposed Dynamic analysis Approach.

behavior. We extracted the corresponding device parameters of Samsung Galaxy S4, Nexus 7 Tablet, Micromax A2 and Karbonn A4 devices.

Property Value	Default AVD	Real Android Device
IMEI	000000000000000	911315462535214
IMSI	310260000000000	925117254763458
Phone Number	15555525554	121314115554
Serial Number	98101430121181100000	54215E52C54525851254
Network	Android	Tmobile
ro.build.id	ICS MR0	IMM76I
ro.build.display.id	sdk-eng 4.0.2 ICS MR0229537 testkeys	TBW592226 8572 V000225
ro.build.version. incremental	229537	TBW592226 8572 V000225
ro.build.version.sdk	19	17
ro.build.version. release	4.0.2	4.2.2
ro.build.date	Wed Nov 23 22:46:18 UTC 2011	2013 01 25 15:53:21 CST
ro.build.date.utc	1322088378	1359100401
ro.build.type	eng	user
ro.build.user	android-build	ccadmin
ro.build.host	vpbs2.mtv.corp.google.com	BUILD14
ro.build.tags	test-keys	test-keys
ro.product.model	sdk	msm7627a
ro.product.brand	generic	qcom
ro.product.name	sdk	msm7627a
ro.product.device	generic	msm7627a
ro.product.board		7x27
ro.board.platform		msm7627a
ro.build.product	generic	msm7627a
ro.build.description	sdk-eng 4.0.2 ICS MR0 229537 testkeys	msm7627a-user 4.0.4 IMM76ITBW592226 8572 V000225 testkeys
ro.build.fingerprint	generic/sdk/generic:4.0.2/ICS MRO/ 229537:eng/test-keys	qcom/msm7627a/msm7627a:4.0.4/IMM76I/ TBW592226 8572 V000225: user/test-keys
net.bt.name	Android	Airtel

Table 7.1: Static parameters: Android AVD and Smartphone.

Furthermore, we replaced the default static emulator values with the real Android device information. This technique had the desired effect on analysis environment aware malware. The `AnServer`, `BgServ` and `Dendroid` malware samples that do not reveal malicious behavior on popular web based services [102, 105, 165] exhibit malicious behavior assuming being run on real Android device.

7.1.2 Analysis tools

The Android SDK has useful and inbuilt analysis tools. In the following, we briefly discuss tools available within the Package:

- **Logcat:** to access system logs and debug the output.
- **Tcpdump:** to intercept the packets sent/received over the network.
- **Monkey:** to Simulate user events.
- **Strace:** to trace System calls.
- **Dumpsys:** to collect the emulator state, snapshot, and components.

7.1.3 Behavioral Analysis

After recording the app actions, we analyze them with `logcat` to detect installed APK, new process spawn and `SMS` sent. We scan the traffic `(.pcap)` files to analyze malicious Uniform Resource Locator (URL) and sensitive data leakage. The System call analysis relate file and network activities. We use Dalvik Dynamic Instrumentation (DDI) to keep track of dynamic operations. The DDI is used to monitor encrypted string operations. The `bind` and `connect` system calls are prominently visible among malicious apps. The following background activities like sending `SMS` and e-mail is considered malicious. The activities prominently present among the malicious apps are:

- Sending confidential device information (International Mobile Equipment Identity (IMEI), International Mobile Subscriber Identity (IMSI), Phone number, etc.).
- Using executable and shell files.
- Modify Permission for any of its files.
- Block app removal after installation.
- Use of System calls prominently used by the malicious APK files.

7.1.4 Network Activity

Android emulator natively supports the network traffic capturing. The proposed approach integrates TCPDump [166] to capture the network traffic into a PCAP file. The bytes sent/received, URL connected to and messages sent to hard coded numbers are monitored to identify suspicious activities. Further, the host connected to, port used to connect, and data sent is identified. The malicious domains are identified by comparison with the URL blacklists. The PCAP content classifies the network activity during the post processing.

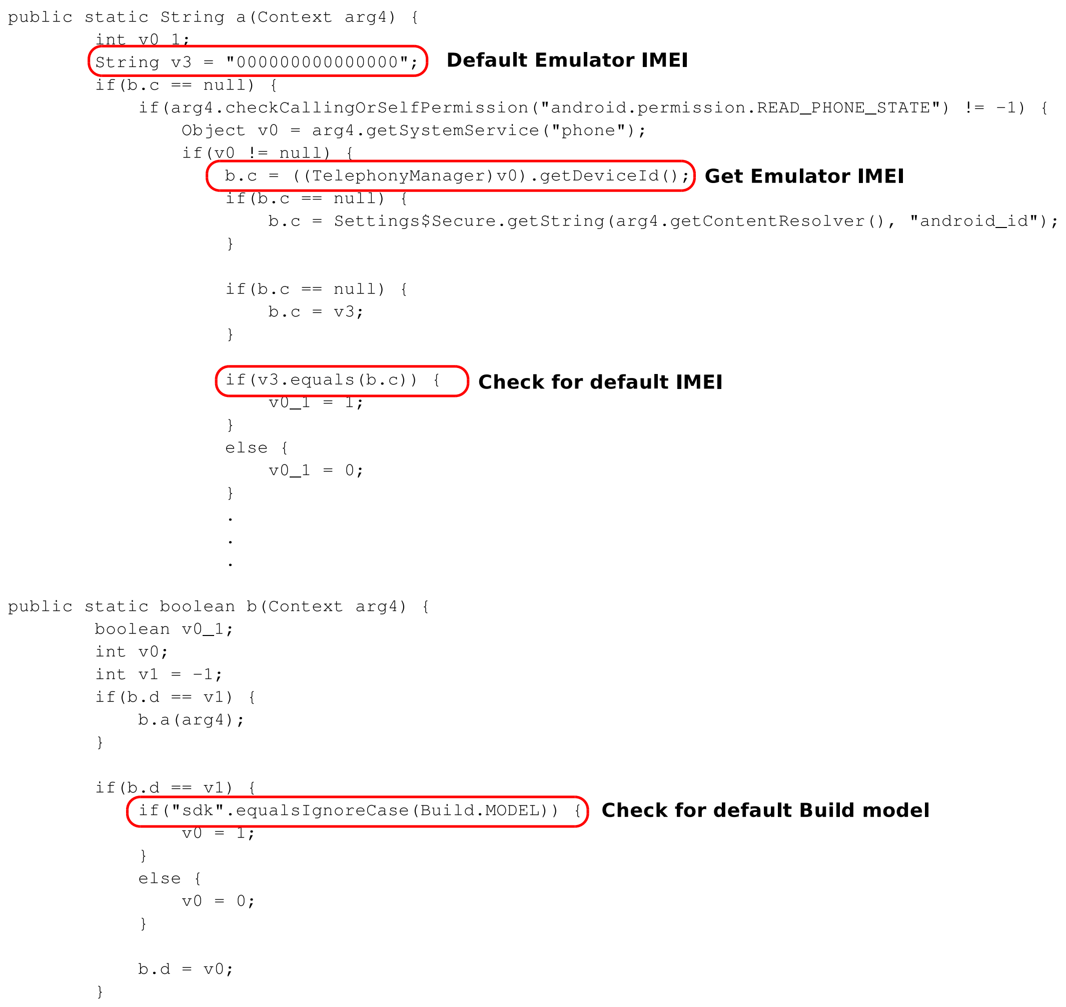
7.2 Experimental Evaluation

The proposed analysis technique is deployed as an Off-Device Linux-based analysis system installed on Intel Pentium Core i7 processor, 16 GB RAM with multiple virtual clones running in parallel.

The dynamic module takes an average 7-12 minute execution time. The execution time limit is 15 minutes. The emulator takes about 50 seconds to reboot the clean state and load the modified virtual device for subsequent execution. The Android Monkey [164] is a part of Android SDK to automate the user gestures during development. We leverage the Monkey to spend an average 5 minutes for app interaction. The post processing techniques need about 2 minutes to extract features from the execution logs. The extracted information is classified with Tree based machine-learning classifiers. The proposed technique is capable of running eight emulation instances in parallel; However, it can be scaled further according to the analysis requirements.

7.2.1 Evaluating Environment aware malware

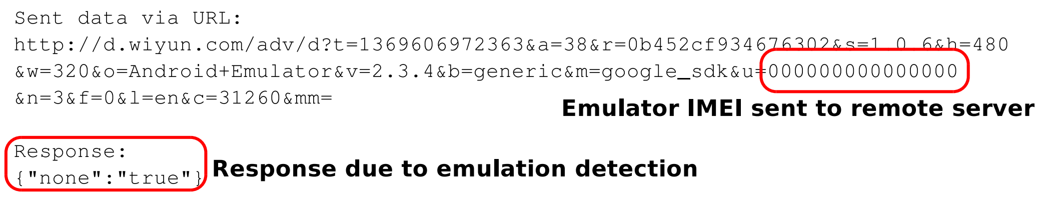
Android malware identifies the emulated or virtual environment and behaves benign to evade the analysis system. Various static and dynamic techniques exist to detect the emulated analysis systems. Figure 7.2 illustrates the Anserver malware code snippet checking the default emulator presence based on `IMEI` and `Build.Model` values. If a malware identifies emulated environment, it behaves benign hiding the malicious payload. The same malware executes inside the modified virtual device and starts sending `IMEI` and confidential user data to the remote server.



```
public static String a(Context arg4) {
        int v0_1;
        String v3 = "000000000000000";
        if(b.c == null) {
            if(arg4.checkCallingOrSelfPermission("android.permission.READ_PHONE_STATE") != -1) {
                Object v0 = arg4.getSystemService("phone");
                if(v0 != null) {
                    b.c = ((TelephonyManager)v0).getDeviceId();
                    if(b.c == null) {
                        b.c = Settings$Secure.getString(arg4.getContentResolver(), "android_id");
                    }

                    if(b.c == null) {
                        b.c = v3;
                    }

                    if(v3.equals(b.c)) {
                        v0_1 = 1;
                    }
                    else {
                        v0_1 = 0;
                    }
                    .
                    .
                    .

public static boolean b(Context arg4) {
        boolean v0_1;
        int v0;
        int v1 = -1;
        if(b.d == v1) {
            b.a(arg4);
        }

        if(b.d == v1) {
            if("sdk".equalsIgnoreCase(Build.MODEL)) {
                v0 = 1;
            }
            else {
                v0 = 0;
            }

            b.d = v0;
        }
```

Figure 7.2: Anserver emulator detection code.

Figure 7.3 illustrates detection of emulated environment by the Anserver, a malicious bot. The default emulator is treated as analysis environment from the parameters listed in Table 7.1

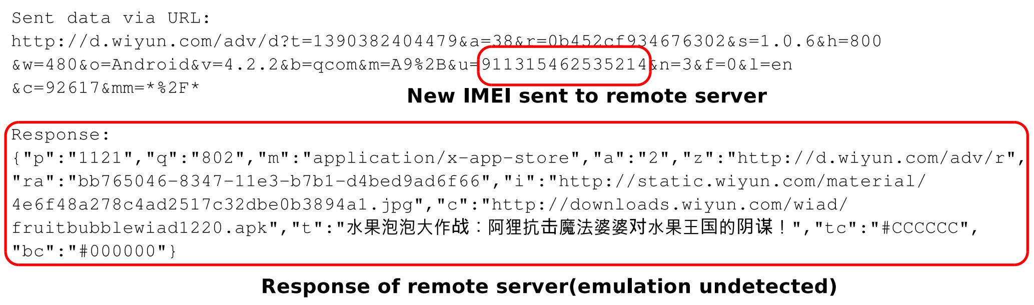


```
Sent data via URL:
http://d.wiyun.com/adv/d?t=1369606972363&a=38&r=0b452cf934676302&s=1.0.6&h=480
&w=320&o=Android+Emulator&v=2.3.4&b=generic&m=google_sdk&u=000000000000000
&n=3&f=0&l=en&c=31260&mm=

Response:
{"none":"true"}
```

Emulator IMEI sent to remote server
Response due to emulation detection

Figure 7.3: Existing frameworks evaded by malware behaving benign.

Figure 7.4 illustrates Anserver revealing hidden malicious behavior assuming the proposed sandbox as a real Android device. The Anserver bot sends the `IMEI` and device information to a remote server, thus revealing malicious behavior.



```
Sent data via URL:
http://d.wiyun.com/adv/d?t=1390382404479&a=38&r=0b452cf934676302&s=1.0.6&h=800
&w=480&o=Android&v=4.2.2&b=qcom&m=A9%2B&u=911315462535214&n=3&f=0&l=en
&c=92617&mm=*%2F*
```

```
Response:
{"p":"1121","q":"802","m":"application/x-app-store","a":"2","z":"http://d.wiyun.com/adv/r",
"ra":"bb765046-8347-11e3-b7b1-d4bed9ad6f66","i":"http://static.wiyun.com/material/
4e6f48a278c4ad2517c32dbe0b3894a1.jpg","c":"http://downloads.wiyun.com/wiad/
fruitbubblewiad1220.apk","t":"水果泡泡大作战：阿狸抗击魔法婆婆对水果王国的阴谋！","tc":"#CCCCCC",
"bc":"#000000"}
```

Figure 7.4: Proposed technique reveals malicious behavior.

7.2.2 Machine Learning Evaluation

We select representative benign Google Play apps and malware from known repositories. These apps are used to train the framework by recording behavioral information. The collected features are trained on Tree based machine learning classifiers. The Random Forest classifier [132] is a nearest neighbor regression predictor and classifier that constructs multiple decision trees. The Random forests forms a strong learner from a group of weak learners [132]. Hence, Tree-based Classifiers are considered for accurate classification. The machine learning model used k-fold cross–validation to discriminate malicious APK from benign.

Table 7.2 illustrates machine learning classification results for 246 analysis environment aware malware and 125 benign apps.

Model Type	Model Name	Correct Prediction (%)	Incorrect Prediction (%)	True Positive (%)	False Negative (%)	True Negative (%)	False Positive (%)
Trees	J48	85.36	14.64	80.17	19.83	90.40	9.60
Trees	Random Forest	86.17	13.82	84.30	15.70	88.00	12.00
Trees	Random Tree	85.36	17.64	83.47	16.53	87.20	12.80
Trees	REPtree	85.36	17.64	80.99	19.01	89.60	10.40

Table 7.2: Classification of environment aware malware.

7.3 Discussions

In this Section, we discuss the following aspects of the experimental evaluation: (i) app stimulation; (ii) system correctness; (iii) performance evaluation; (iv) environment-aware malware detection; and (v) scalability analysis.

7.3.1 App Stimulation effects

The system stimulates the app to respond the interaction with analysis system. To evaluate the stimulation effect, we selected 172 Google Play and 165 malicious apps. We forcefully invoke the main activity component. In the first stage, individual stimulation techniques (invoke only; main activity, monkey tool, app stimulation and DDI hooking) were applied. The effect of different stimulation is illustrated in Figure 7.5.

The first bar illustrates the invoking main activity. The second stimulus is based on monkey gestures and main activity. The third bar displays the response to differently implemented stimulation techniques. The last bar illustrates the effect of combined stimulation techniques. We can see the effect of combined stimulus has high code coverage. For example, random clicks generated by the Android Monkey triggers SMS-sending activities. However, services are triggered with the service iterators. It is also interesting to note that combination of multiple techniques has high coverage as compared to a single method. Hence, the integration of the synergy of static and dynamic analysis is justified.

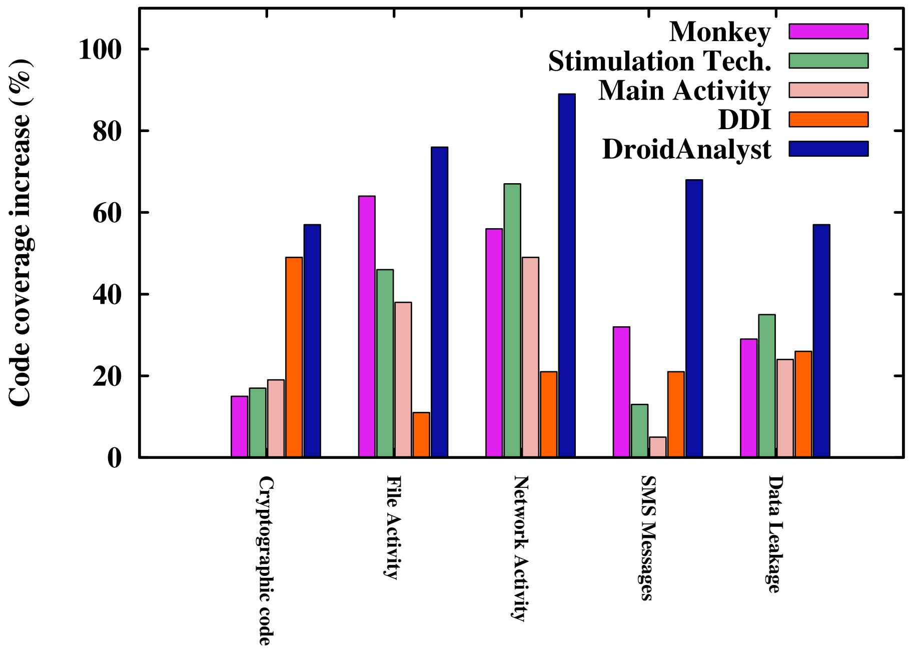


Figure 7.5: Comparison of Stimulation techniques.

7.3.2 System correctness

The system correctness methodology ensures that the logs of an APK are generated when such action deemed to have been performed. We select random 10 samples from known families performing varied malicious activities. Anserverbot malware checks for `IMEI` and `Model.build` to verify the presence of analysis environment. If the app detects emulator based on static properties, it hides the malicious behavior. If the malicious APK identifies random numeric values, it sends the `IMEI` to a remote server.

`FAkeInstaller` malware sends premium rate `SMS` messages without user consent. `RootSmart` malware employs root exploits and executes native calls to exploit the device. `FakeInst` and `AdSMS` sends premium `SMS` messages apart from sending the `IMEI` and `IMSI` numbers to the remote server. The `TapSnake` malware family misuse the user location for targeted advertisements. The above information is verified and reported by the leading commercial anti-malware and malware researchers. The reason for the improved effect is due to synergic use of static and dynamic analysis techniques.

7.3.3 Scalability

The third parameter tests the scalability. The scalability is evaluated against apps crawled between February 2013 and December 2014. A total 47,342 apps were crawled from Google Play, Anzhi and other third party Asian markets; 26,469 malicious are downloaded from *VirusShare*, *contagiominidump* and other third party markets. We also received 312 new malware samples based on user uploads. Out of the total, we randomly selected 6,743 benign and 2,786 from the malware dataset.

We performed analysis on intel core i7 8 GB memory. 217 Google Play apps were labeled malicious with the proposed analysis technique. These apps were already labeled benign by the commercial anti-malware. More samples belonged to `FakeInstaller`, `AnServer`, `Kmin` and `FakePlayer` families using premium-rate SMS service or user data ex-filtration.

7.3.4 Scope for improvements

The absence of device sensors can be used to detect the emulated environment. Hence, it is a challenge for malware authors to adopt smart techniques to evade the analysis environment. To reduce the false alarm, we would develop an on-device analysis system with important features to improve the real time malware detection accuracy.

7.4 Comparison with Existing Work

Table 7.3 illustrates comparative analysis of known dynamic analysis frameworks with our proposal. We compare the analysis techniques based on (1) scalability; (2) resource consumption; (3) API hooking; (4) encrypted text monitoring; (5) file operations; and (6) sensitive data ex-filtration.

Property	**AASandbox [121]**	**Andromaly [58]**	**Apps Playground [165]**	**Droidbox [106]**	**Andrubis [118]**	**Proposed Approach**
Scalability	✔	✔				✔
Resource use		✔				✔
API Hooking			✔	✔	✔	✔
Logcat analysis						✔
System call	✔					✔
analysis aware malware			✔	✔	✔	✔
Data exfiltration			✔	✔	✔	✔
SMS misuse			✔			✔
Traffic analysis				✔	✔	✔
File Operations monitoring				✔	✔	✔

Table 7.3: Comparing Proposed approach.

Droidbox and TaintDroid form base of other existing dynamic frameworks such as Andrubis, Apps Playground, and SmartDroid. Our proposal modifies the default Android emulator to analyze the environment-aware malware without modifying the Android OS. CopperDroid [167] is a system call based analysis framework to monitor inter-process communication using Virtual Machine Introspection (VMI) technique. The proposed technique evaluates 1,260 samples from 49 Android Genome [2] malware repository. The Proposed framework utilizes `strace` to record system calls. We utilize Dalvik Dynamic Instrumentation (DDI) [168] to monitor the cryptographic and sensitive API calls. Andrubis [118] is a web-based service for analyzing malicious apps using both static and dynamic analysis.

Petsas et al. [117] compare advanced malware evading the virtual/emulated environment. The authors patched existing malware apps with *anti-analysis* features to evade the existing analysis techniques [167, 118, 121, 122]. The proposed Sandbox addresses the concern to propose environment-aware malware detection framework that uncovers evolving threats. We have compared the proposed analysis technique based on 10 parameters to compare the relevant dynamic analysis techniques illustrated in 7.3.

7.5 Summary

In this Chapter, we discuss the intricacies of a user-driven, dynamic analysis framework that uncovers analysis environment aware Android malware. The proposed technique employs an improved Android Virtual Device based dynamic analysis that successfully reveals hidden behavior prevalent among the evolving Android malware. The dynamic analysis module counters the obfuscated and encrypted payloads to uncover the advanced malware threats without modifications to the Android framework.

Chapter 8

Conclusions and Future directions

8.1 Conclusions

In this Thesis, we present malware analysis and detection techniques as a synergic combination of static and dynamic analysis. We propose multiple novel analysis and detection techniques to detect "single malicious app" which complement each other to improve analysis coverage. In the first step, we analyze the Android manifest file permissions and map them with their actual usage in the Dalvik bytecode. This technique identifies over-privileged apps that can be misused by the malware authors.

In the second step, we propose AndroSimilar, a robust statistical signature to identify repackaged malware and unseen variants of known Android malware families. The proposed AndroSimilar utilizes statistically robust features generated using SDHash algorithm to create variable-length signatures for detection of repackaged malware and unseen variants of known malware. The proposed methodology is robust against trivial string encryption, method renaming, junk method insertion control flow obfuscation techniques. In fact, we were able to identify repackaged applications evading the existing anti-malware techniques.

However, the recent emerging malware employs covert behavior to execute malicious functionality such as sending SMS, dialing premium-rate numbers, recording audio/video, taking pictures without explicit user consent. We identify such *covert actions* as *sensitive feature misuse*. To identify such hidden malicious action, we proposed *CONFIDA*, an inter-component communication-based detector for identifying sensitive feature misuse. The proposed approach identifies sensitive functionality necessitating explicit user intervention. We generate Dalvik bytecode ICC based component-interaction graph to identify the misuse of sensitive Android API. We evaluated 951 malicious and 1157 benign apps with classification accuracy 2.3% false negative rate and 2.1% false positive rate, superior to the existing approaches in the literature. However, obfuscation, dynamic class loading and reflection API limits the static analysis.

To identify the resilience of proposed techniques against obfuscation, in the next step we proposed an automatic obfuscator "DroidSHornet". Furthermore, we evaluated the Dalvik bytecode obfuscator resilience against Android bytecode transformations. The top rated anti-malware are vulnerable against permutations of control, data and layout obfuscation. The proposed analysis technique is superior in comparison to the existing analysis techniques presented in [46, 47]. Our proposal AndroSimilar outperforms the existing anti-malware techniques evaded by the obfuscated malware. The limitations of existing static analysis approach motivated the proposal for a complementary dynamic analysis to aid and improve the code coverage.

The evolving malware have inbuilt capability to identify the emulated analysis environment. Once the app identifies the analysis sandbox or virtual environment it behaves benign. To uncover such environment aware malware, we propose a framework with modified static emulator properties and enrich the virtual device with essential user information. The experimental evaluation shows a marked improvement in analysis efficacy due to the synergic use of static and dynamic analysis techniques in comparison to the state-of-the-art "single malware" app detection.

The proposed techniques reduce false alarms and improves the detection capabilities. The implementation of multiple, synergic and proactive analysis techniques presented in the Thesis allow defenders to stay ahead of malware authors in the attack defense race. The analysis techniques have been integrated into DroidAnalyst, an app analysis engine briefly discussed in Appendix B. Similarly, additional techniques can be integrated into APK analysis to improve the code coverage.

8.2 Future Work

The are many future directions to to extend our work. We envision to extend the work presented in this Thesis in several directions. The analysis techniques have been applied for detection of a "single malicious app". We envision that the proposed techniques can be applied to detect "colluding apps", a new paradigm in malware research. Furthermore, obfuscation and code protection techniques on Android pose an important challenge for the static and dynamic program analysis techniques.

The proposed techniques can be implemented on other popular mobile platforms. Synergic use of static and dynamic analysis techniques ensures the improved analysis and detection coverage. The dynamic analysis can complement the static analysis if reflection code, dynamic code loading, or presence of native code. There are few additional extensions for future work:

1. Simulate sensor features to strengthen the default emulator.
2. Facilitate lightweight, on-device malware app analysis.
3. Detection of malware samples performing audio/video recordings and sensor based malicious behaviour.
4. Perform API monitoring using Dalvik bytecode hooking technique.
5. Integrate the proposed methodology as a real-time malware app detection framework.

Finally, we plan to extend the evaluation and interpretations of multiple analysis techniques towards improving the existing analysis system. The synergic use of complementary techniques will aid the human analyst attain detailed insight to analyze multiple apps and rationale behind the false alarms.

Appendix A

Android Analysis Tools and Techniques

A.1 Android Threats

AOSP is committed to a secure Android Platform. However, it can be attacked with social-engineering tricks. Once the app is installed, it may create undesirable consequences on the device security. Following is the list of malicious activities that have been reported or can be employed in subsequent Android versions.

1. *Privilege escalation* attacks were leveraged by exploiting publicly available Android kernel vulnerabilities to gain root access of the device [169]. Android exported components might be used to obtain access to the dangerous permissions.

2. *Privacy leakage or sensitive user information theft* occurs when users grant dangerous permissions to malicious apps and unknowingly allows access to sensitive data and ex-filtrate them without user knowledge and consent.

3. Malicious apps can also *spy* on the users by monitoring the voice calls, SMS/MMS, bank mTANs, recording audio/video without user knowledge or consent.

4. Malicious apps can earn money by making calls or subscribe to premium rate number SMS without the user knowledge or consent.

5. Compromise the device to act as a *Bot* and remotely control it through a server.

6. *Aggressive ad campaigns* may entice users to download Potentially Unwanted Apps (PUA) or malware apps [170].

7. *Colluding* attack happens when a set of apps, signed with the same certificate, gets installed on a device. These apps would share Unique IDentifier (UID) with each other. Also, any dangerous permission(s) requested by one app can be shared by the colluding malware. Collectively, these apps perform malicious activities, whereas, their individual functionality is benign. For example, an app with `READ_SMS` permission can read SMSes and ask the colluding partner with `INTERNET` permission to ex-filtrate the sensitive information to a remote server.

8. *DoS* attack can happen when the app(s) overused already limited CPU, memory, battery and bandwidth resources and restrains the users executing standard functions.

A.2 Fragmentation Problems

Android Open Source Project led by Google upgrades and maintains the OS code. A Patch, an update or major upgrade distribution is the responsibility of OEM. The OEM branches out updated versions of the OS and customize them. In some countries, the wireless carriers customize the OEM OS to suit their requirements. Such an update chain takes months before the patch reaches the end-users. This phenomenon is called *Fragmentation*, where different versions of Android remain scattered due to the unavailability of updates. Specifically, handsets with older and unpatched versions remain vulnerable to the known exploits.

A.2.1 Native Code Execution

Android allows native code execution through libraries implemented in C/C++ using Native Development Kit (NDK). Even though native code executes outside Dalvik VM, it is sandboxed through user-id/group-id(s) combination. However, the native code has the potential to perform privilege escalation by exploiting platform vulnerabilities [34], [36, 171, 172, 35, 173]. The attacks has been demonstrated in the recent past [29].

A.3 Android Platform Security enhancements

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

1. Android prevented stack buffer and the integer overflow in the OS version 1.5. In version 2.3, Android fixed `string format` vulnerabilities and added hardware based No eXecute (NX) support to stop the execution of code in stack and heap [1].

2. In Android 4.0 Address Space Layout Randomization (ALSR) was added to prevent the *return-to-libc and memory related attacks* [1].

3. Information ex-filtration by connecting the device to a PC using the Android Debug Bridge (ADB). The ADB is developed as a debugging tool. However, it permits app installation, read system and partitions even when the device is locked but connected to PC. To prevent such unauthorized access, Android 4.2.2 authenticates an ADB connection with an RSA keypair [174]. The user response is prompted on the device screen if the ADB connection accesses the device. Thus, if the device is locked, an attacker would not be able to gain the control.

4. To prevent the malware from silently sending premium-rate SMS messages, Android 4.2 introduced an additional notification feature to prompt the user before a user app sends an SMS [175].

5. Android proposed a significant capability addition to the version 4.2 (API version 17). This version permits Multiple Users (MU) on a single device [109]. The Restricted Profile (RP) access capability was introduced added in Android 4.3 (API version 18) in July 2013. These modifications were placed keeping in mind the usage of sharable mobile devices such as tablets to provide private space to multiple users on a single mobile device.

For each user, a separate account, the user selected apps, custom settings, personal data and file space. This capability enables the multiple users share a single device. In the MU scenario, the primary account is the owner of the mobile. Using the device settings, the device owner creates additional MU. The first user is permitted to create, modify or delete the additional user.

6. Android 4.3 removed the `setuid()/setgid()` programs [174] as they were vulnerable to the root exploits.

7. Android 4.3 experimented with Security Enhanced Linux (SELinux) to provide the enhanced security [176]. Android 4.4 introduced SELinux with enforcing mode for multiple root processes. SELinux imposed Mandatory Access Control (MAC) policies in place of the traditional Discretionary Access Control (DAC). In DAC, resource owner decides which other interested subjects can access it. However, in MAC the system (not the users) authorizes the subject to access a particular resource. Thus, MAC has the potential to prevent the malicious activity(s) even if the root access of the device is compromised. Thus, MAC substantially reduces the effect of kernel-level privilege escalation attacks.

A.3.1 Third-party Security Enhancements

Many independent Android security enhancements have been proposed [177, 178, 179, 180]. These mechanisms allow an organization to create fine-grained security policies for their employee devices. Contextual information such as device location, app permissions, and inter-app communication can be monitored and verified against the already declared policies. In this chapter, we investigate the Android security, malware issues and defense techniques.

A.3.2 Reverse-Engineering Tools

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

1. *apktool* [42] can decode the binary content of an APK into nearly original form in the project-like directory structure. It disassembles the binary resources and converts bytecode within `classes.dex` and smali [161] bytecode. It can also repackage it back into an APK. This tool is one of the best open-source reverse-engineering tools.

2. *dex2jar* [83] is a disassembler to parse both the `.dex` and optimized `dex` file, providing a light-weight API to access it. *dex2jar* can also convert `dex` to a `jar` file, by re-targeting the Dalvik bytecode into Java bytecode, for further manipulation. Moreover, it can also re-assemble the `jar` into a `.dex` after the modifications.

3. *Dare* [181] project aims at re-targeting Dalvik bytecode within `classes.dex` to traditional `.class` files using strong type inference algorithm. This `.class` files can be further analyzed using a range of traditional techniques developed for Java applications, including the decompilers. Octeau et al. [80] demonstrated that *Dare* is 40% more accurate than *dex2jar*.

4. *Dedexer* [182] disassembles the `classes.dex` into Jasmin-like syntax and creates a separate file for each class maintaining the package directory structure for easy reading and manipulation. However, unlike the *apktool*, it cannot re-assemble the dis-assembled intermediate class files.

5. *JEB* [108] is a leading professional Android reverse-engineering software available on Windows, Linux, and Macintosh platforms. It is a GUI-based interactive decompiler analyzes the reversed malware app content. App information such as manifest, resources, certificates, literal strings can be examined in Java source by providing an easy navigation through the cross-references. JEB converts the Dalvik bytecode to Java source from the Dalvik bytecode. Exceptionally, JEB can also de-obfuscate Dalvik bytecode to make disassembled code more readable in comparison to its counterparts [83, 42]. JEB supports Python scripts or plugins by allowing access to the decompiled Java code Abstract Syntax Tree (AST) through API. This feature is helpful in automating the custom analysis. According to us, it is the best reverse-engineering tool so far.

A.4 Android Analysis Tools

In this Section, we discuss the most popular open source static and dynamic analysis tools available on Android platform.

A.4.1 Androguard

Figure A.1 illustrates Androguard [52] an open-source, static analysis tool can reverse engineer to disassemble and decompile Android apps. It generates the control flow graphs for each method and provides access through Python-API on the command line and graphic interface. Androguard NCD approach finds similarities and differences of two suspected clones reliably, which is also helpful to detect repackaged apps. It provides Python APIs to access the disassembled resources and static analysis structures like basic-blocks, control-flow and instructions of an APK. An analyst can develop his static analysis framework using the Python APIs. Following are some of the features explained below.

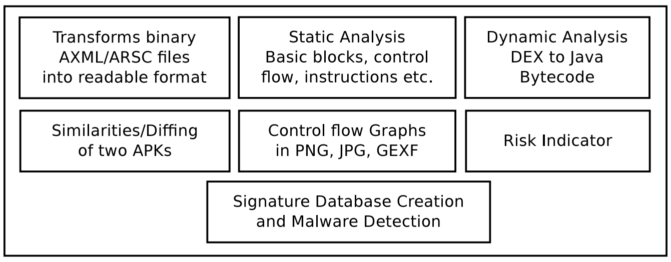

Transforms binary AXML/ARSC files into readable format
Static Analysis Basic blocks, control flow, instructions etc.
Dynamic Analysis DEX to Java Bytecode
Similarities/Diffing of two APKs
Control flow Graphs in PNG, JPG, GEXF
Risk Indicator
Signature Database Creation and Malware Detection

Figure A.1: Androguard features.

A.4.1.1 App code similarity

Androguard finds similarities between two apps by calculating Normalized Compression Distance between each method pairs and calculates a similarity score between 0-100, where 100 means identical apps. It displays IDENTICAL, SIMILAR, NEW, DELETED and SKIPPED methods of the two suspected clones. In the same way, it displays differences between two methods by comparing each basic blocks pairs. More specifically, to calculate differences between two similar methods, it first converts each unique instruction in the basic block into a string. Then, it applies Longest Common Subsequence algorithm on these strings of two basic blocks to find differences between them [183].

A.4.1.2 Risk Indicator

Risk Indicator calculates the 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.

A.4.1.3 Signature of Malicious Apps

Androguard manages a database of signatures and provides an interface to add/remove signatures to/from the database in JSON format. It contains a name (or family name), set of sub-signatures and a Boolean formula to mix different sub-signatures.

A.4.2 Andromaly

In [58], Shabtai et al. have proposed a light-weight Android malware detection system based on machine learning approach. It performs real-time monitoring of CPU usage, transferred data, the number of active processes and battery usage.

As illustrated in Figure A.2, Andromaly has four main components:

- *Feature Extractors:* Collects the feature metric by communicating with Android kernel and application framework. Feature Extractors are triggered to collect new feature measurements by the feature manager. Feature Manager may also perform some pre-processing on the raw feature data.
- *Processor:* It is an analysis and detection unit. It receives the feature vectors from Main Service, analyze them and perform the threat assessment and pass it on to Threat Weighting Unit (TWU). The Processors can be rule-based, knowledge-based classifiers or anomaly detectors employing machine learning methods. TWU applies ensemble algorithm on the analysis results received

from all the processors to derive a final decision on the device infection. Alert Manager smoothes the results to reduce the false alarms.

- *Main Service:* The main service co-ordinates feature collection, malware detection and alert process. It handles requesting new feature measurements, sending new feature metrics to the processors and receives final recommendations from the alert manager. Loggers can log information for debugging, calibration and experimentation. Configuration Manager configures an app (for example, active processors, alert threshold, and sampling intervals). The Processor Manager activates/de-activates the processors. The Operation Mode Manager switches the application from one mode to another for the purpose of feature extraction. This change in operation modes occurs due to change in resource levels.

- *Graphical User Interface (GUI):* The GUI configures application parameters, activates/deactivates the app, sends threat alerts, explores collected data. Experiments were carried out using few categories of artificial malware, thus working model needs testing by real malware.

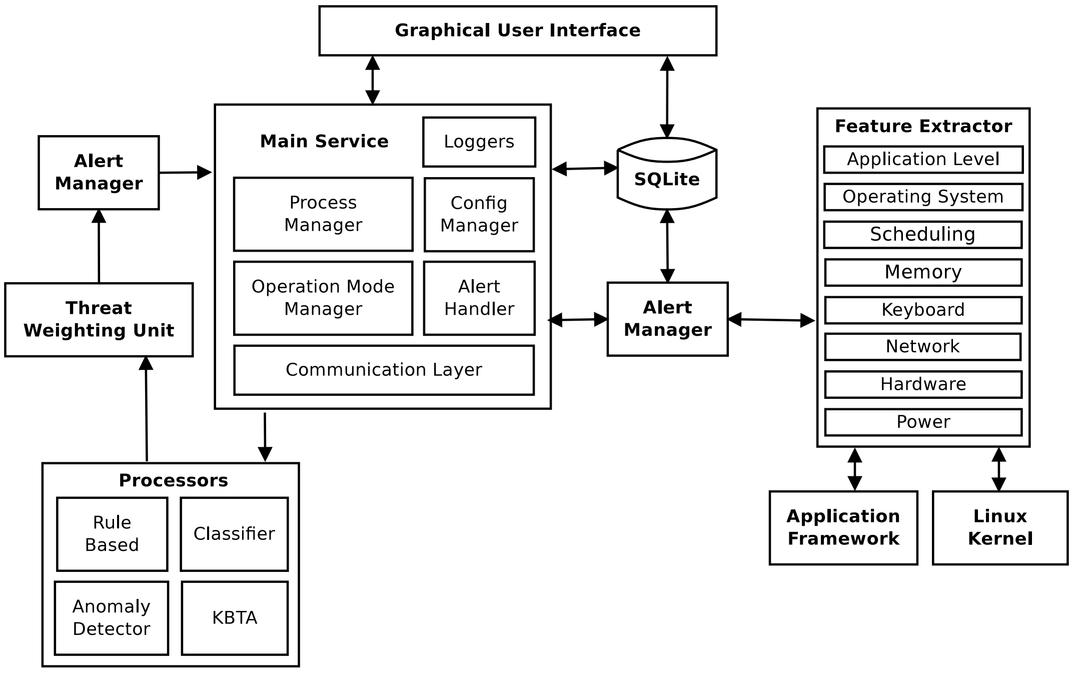


Figure A.2: Architecture of Andromaly.

A.4.3 Andrubis

Andrubis [102] is a web-based malware analysis platform, built on top of Droidbox [106], TaintDroid [92], apktool [42] and Androguard [52]. Users can submit suspicious apps through the web-based interface. After analyzing the app at the remote server, Andrubis returns analysis reports. Andrubis also provides app behavior rating between 0-10, where 0 indicates benign, and 10 specifies high malicious rating.

To study the Andrubis functionality, a custom developed SMS botnet was uploaded on the Andrubis web service. This research prototype rated custom SMS bot with a score 9.9/10. However, none of the commercial anti-malware at the VirusTotal portal detect the hidden malware. Furthermore, Vidas et al. [89] reports many static anti-analysis techniques evading the Andrubis web service.

A.4.4 APKInspector

APKInspector [103] is a full-fledged Android static analysis tool, consisting *Ded* [184], *smali/baksmali* [161], *apktool* [42] and *Androguard* [52]. It provides a rich GUI and has following features:

- App meta-data
- Analysis of sensitive permissions
- Displays Dalvik bytecode and Java source code
- Displays control-flow graph
- Displays call-graph, displaying call-in and call-out structures
- Static instrumentation support by allowing modification to the *smali* code

A.4.5 Aurasium

Aurasium [104] is a powerful technique that takes control of the execution of apps, by enforcing arbitrary runtime security policies. To be able to do that, Aurasium repackages the Android apps with the policy enforcement module. Aurasium Security Manager component can apply policies on the individual and multiple apps. Any security and privacy violations are reported to the user. Thus, it eliminates the need for manipulating Android OS to monitor app behavior.

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

A.4.6 Bouncer

Google protects the Google Play with its own anti-malware known by the name Bouncer. The Bouncer is a virtual machine based dynamic analysis platform to test the uploaded third-party developer apps, before availing them to the users for download. It executes app to look for any malicious behavior and also compares it against previously analyzed malicious apps. The internal functioning documentation is not available with the researchers. However, Oberheide et al. [88] fingerprinted the Bouncer environment with a custom command and control app. The authors reported that, the dynamic code loading techniques can evade the Bouncer [19] scrutiny.

A.4.7 CopperDroid

Reina et al. proposed CopperDroid [90], an Android based system call-centric Virtual Machine Introspection interface. To address the path coverage problem, they supported the stimulation of events as per the specification in the app manifest. The authors reported experimental evaluation regarding accuracy of the proposed detection approach. They have also provided a web interface for other users to analyze apps [105]. However, Vidas et al. [89] demonstrate the identification of CopperDroid's virtual environment by employing advanced anti-analysis techniques.

A.4.8 Crowdroid

Crowdroid [93] is a behavior-based malware detection system. It has two components, a crowd-sourcing app that need to be installed on the mobile 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*, an on-device system utility collects the system-call details. The application log file consists of basic device information, a list of installed applications and behavioral data. At the remote server, this data is processed to create feature vectors that 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.

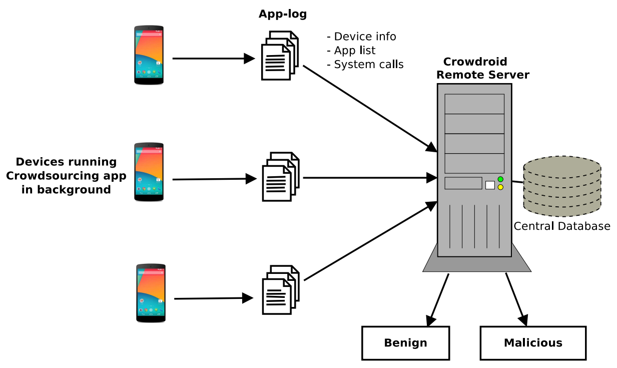


Figure A.3: Crowdroid Architecture.

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 a significant difference in system calls, which can help in detection for the same. However, it also suffers from false-positives, as demonstrated by authors using *Monkey Jump2*, an app with *HongTouTou* malware.

The crowd-sourcing app must always be available for monitoring which drains the available resources. The technique is yet to be tested on the known malware families.

A.4.9 Droidbox

Droidbox [106], illustrated in Figure A.4 is a dynamic analysis tool developed on top of TaintDroid [92]. It modifies the Android framework for API call analysis.

Figure A.4 displays the static and dynamic analysis operations. App analysis begins with the static-pre-checking, which includes parsing permissions, activities, and receivers. The app under analysis is executed in a virtual analysis environment. Taint-analysis involves labeling (tainting) private and sensitive data that propagates through the program variables, files, and interprocess communication.

The Taint-analysis keeps track of tainted data that leaves the system from the network, file(s) or SMS. API monitoring involves API logging with its parameters and return values. The results consist the following parameters:

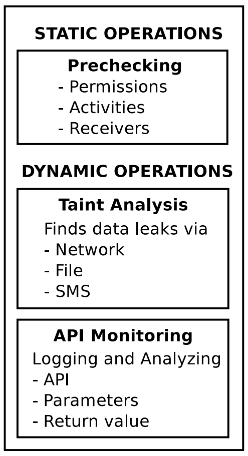


Figure A.4: Droidbox features.

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

- Treemap for similarity analysis

Limitation: Droidbox can only monitor the tasks performed within the Android Framework. If the native code leaks the sensitive data, the existing system cannot detect and hence, the user data is ex-filtrated.

A.4.10 DroidMOSS

DroidMOSS [11] is an app repackaging detection prototype employing semantic file similarity measures. More specifically, it extracts the DEX opcode sequence of an app and generates a signature fuzzy hashing [142] signature from the opcode. It also adds developer certificate information, mapped into a unique 32-bit identifier in the signature. Suspected app features are verified against the original apps using the edit-distance algorithm to identify the similarity score. The proposed approach is discussed and illustrated in Figure A.5.

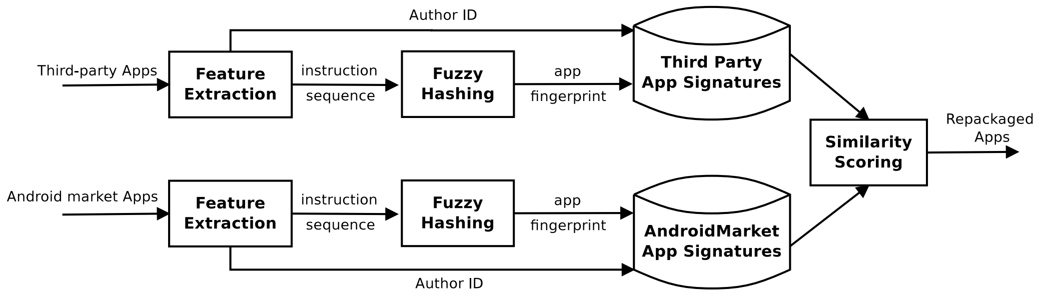


Figure A.5: DroidMOSS Methodology.

Intuition behind DroidMOSS using the opcodes feature is, it might be easy for adversaries to modify operands, but very hard to change the actual opcodes [11]. This approach has several disadvantages. First, it only considers DEX bytecode, ignoring the native code and app resources. Second, the opcode sequence does not consist high-level semantic information and hence generates false negatives. The smart adversary can evade this technique using code transformation techniques such as inserting junk bytecode, restructure methods and alter control flow to evade the DroidMOSS prototype.

A.4.11 DroidScope

DroidScope [100] is a Virtual Machine Introspection (VMI) Android OS based dynamic analysis framework. Unlike other dynamic analysis platforms, it stays out of the emulator and monitors the OS and Dalvik semantics. Hence, even the privilege escalation attacks on the Android kernel can be detected. It also makes the attackers task of disrupting analysis difficult. DroidScope is built upon QEMU emulator with a rich set of APIs to customize the malware analysis prototype. Android malware families DroidKungFu and DroidDream were analyzed and detected with this technique. However, DroidScope's effectiveness against other malware families remains to be tested.

A.4.12 Drozer

Drozer [107] is a comprehensive attack and security assessment framework for Android devices, available as an open-source and a professional version. It allows security enforcement agencies to exploit Android devices and identify vulnerabilities of the Android OS. Figure A.6 displays the Drozer functionality. Following features are supported by the Drozer:

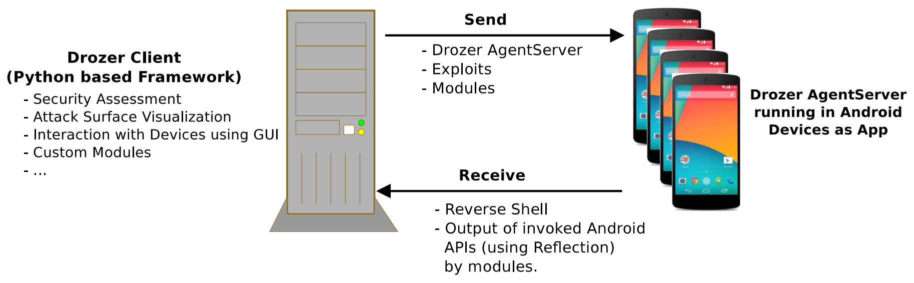

Drozer Client
(Python based Framework)
- Security Assessment
- Attack Surface Visualization
- Interaction with Devices using GUI
- Custom Modules
- ...
Send
- Drozer AgentServer
- Exploits
- Modules
Receive
- Reverse Shell
- Output of invoked Android APIs (using Reflection) by modules.
Drozer AgentServer running in Android Devices as App

Figure A.6: Working of Drozer.

- It installs an Agent app on the devices which executes exploitation modules using Java Reflection API. At server-side, one can create their custom modules in Python and send it to Agent app to perform exploitation.
- It can interact with the Dalvik VM to discover installed packages and related app components. It also allows interaction with the app components like services, content providers and broadcast receivers to identify vulnerabilities. Drozer creates a remote shell to control the device.

- It is capable of generating known exploits taking advantage of the already known rooting vulnerabilities.

A.4.13 Kirin

In [67], authors proposed a security policy enforcement mechanism, *Kirin*, an on-device app vetting framework. Kirin defines a set of rules based on the combination of certain dangerous permissions requested by the app. If an app fails to satisfy the Kirin security rules, installation is prevented.

A.4.14 TaintDroid

TaintDroid [92] extends the Android platform to track the privacy-sensitive information leakage in the third-party developer apps. The sensitive data is automatically tainted (or labeled) to keep track whether the labeled data leaves the device. When the confidential data leaves the system, TaintDroid records the label of the particular information and the app that sent the data along with its destination address. Figure A.7 illustrates the concept. Taint Propagation has granularity at; 1) Variable-level, 2) Method-level, 3) Message-level and 4) File-level. Variable-level tracking uses variable semantics for necessary context to avoid taint propagation. In message-level tracking, the taint on messages is tracked to prevent IPC overhead.

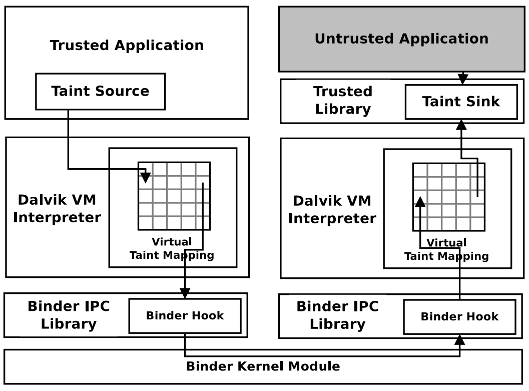


Figure A.7: Taint propagation in TaintDroid.

Finally, the file-level tracing ensures the integrity of file-access activities by checking whether taint markings is retained. First, the information of the trusted app is labeled according to its context. A native method interfaces with the Dalvik VM interpreter to store the taint markings in a virtual paint map.

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

Appendix B

DroidAnalyst: Malware Analysis and Detection Engine

B.1 DroidAnalyst

The appendix gives a brief overview of DroidAnalyst, an automated app vetting and malware analysis framework that integrates the synergy of static and dynamic analysis. DroidAnalyst generates a unified analysis model that combines the strength of the complementary approaches with multiple detection methods. The APK analysis engine employs state-of-the-art static and dynamic analysis techniques displays results in a human readable format. Thus, it helps determine human analyst to take an informed decision about analysis results of a suspect APK file.

B.2 DroidAnalyst: Brief overview

The DroidAnalyst can be used by the malware analysts in a local network by following the standard installation procedure.

- For creating a new user, Click on **Sign Up** button and fill the necessary details. After successful submission, a verification link is sent to the email address provided.

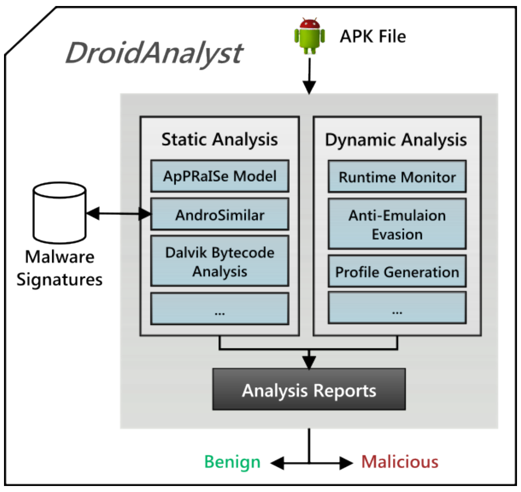


Figure B.1: DroidAnalyst Framework

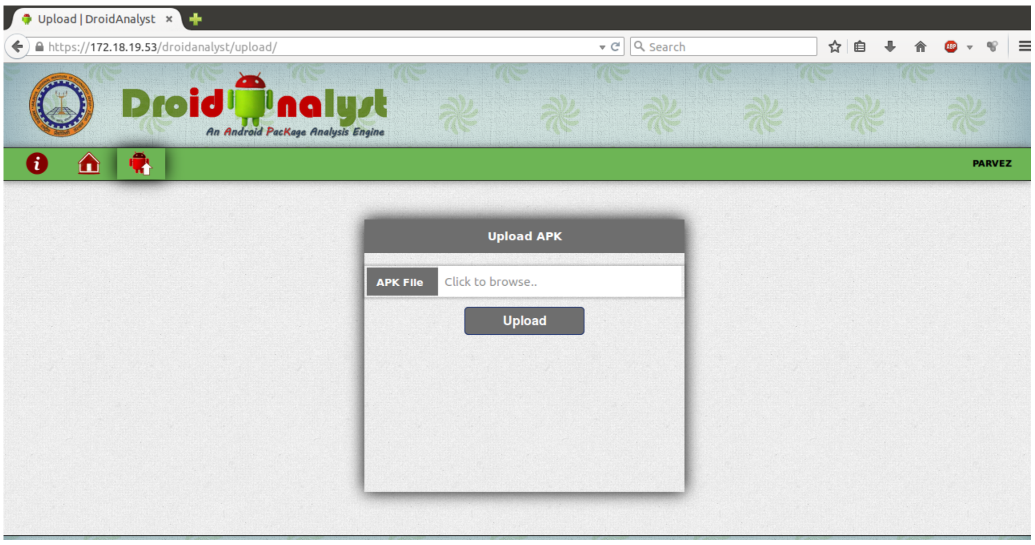
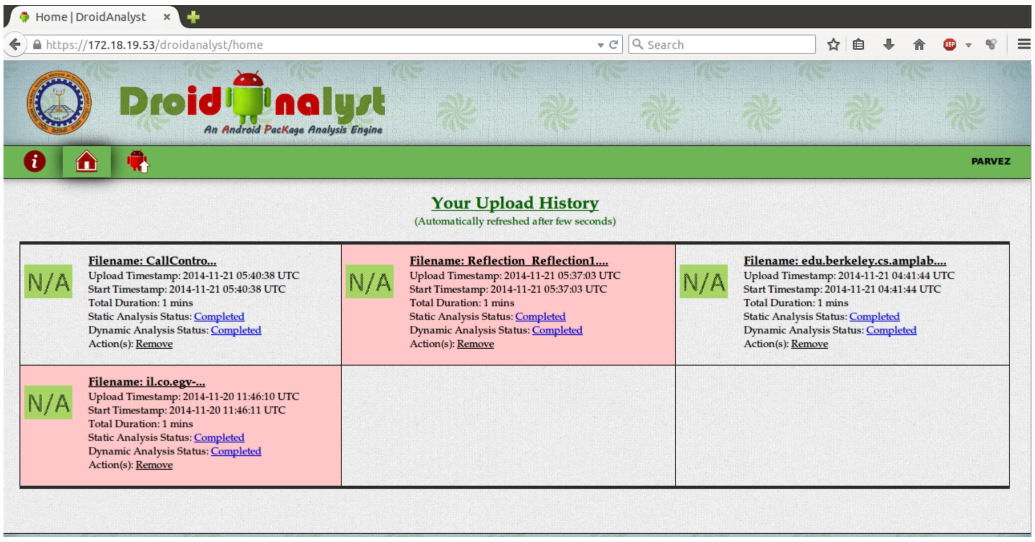


Figure B.2: Uploading .apk file for analysis

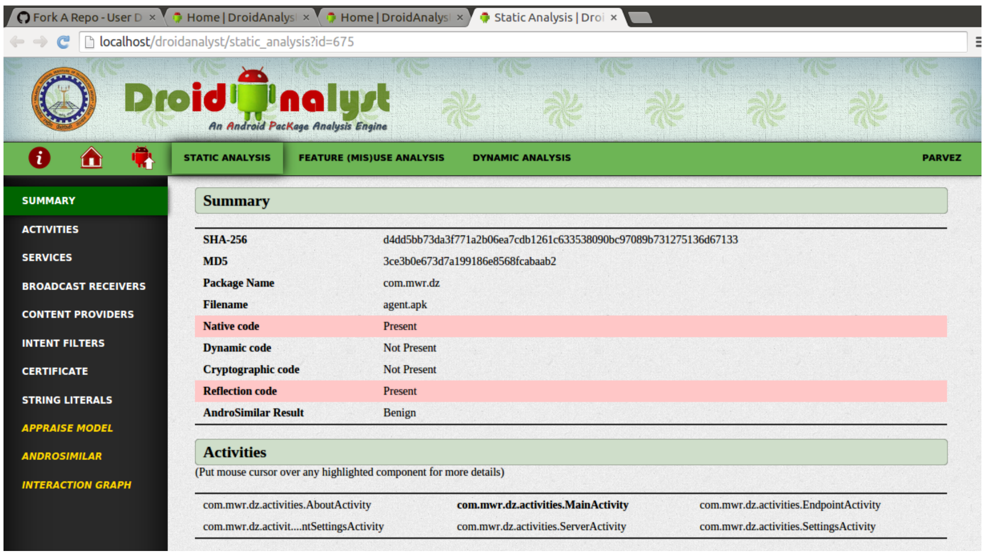
- Click the verification link to start using DroidAnalyst.
- Login to your account and upload the **.apk** file.
- At present we allow the maximum size of **.apk** as 20 MB.
- Once the file is uploaded successfully; the framework adds the sample to the service queue, internally managed by DroidAnalyst. User Home tab illustrates the analysis status as shown in Figure: B.3
- Possible status:
  - **Not Started**: Analysis yet to begin.
  - **In Progress**: Analysis has started but not yet completed.


Home | DroidAnalyst
https://172.18.19.53/droidanalyst/home
Search
DroidAnalyst
An Android PacKage Analysis Engine
PARVEZ

Your Upload History
(Automatically refreshed after few seconds)

N/A **Filename: CallContro...** Upload Timestamp: 2014-11-21 05:40:38 UTC Start Timestamp: 2014-11-21 05:40:38 UTC Total Duration: 1 mins Static Analysis Status: Completed Dynamic Analysis Status: Completed Action(s): Remove	N/A **Filename: Reflection_Reflection1....** Upload Timestamp: 2014-11-21 05:37:03 UTC Start Timestamp: 2014-11-21 05:37:03 UTC Total Duration: 1 mins Static Analysis Status: Completed Dynamic Analysis Status: Completed Action(s): Remove	N/A **Filename: edu.berkeley.cs.amplab....** Upload Timestamp: 2014-11-21 04:41:44 UTC Start Timestamp: 2014-11-21 04:41:44 UTC Total Duration: 1 mins Static Analysis Status: Completed Dynamic Analysis Status: Completed Action(s): Remove
N/A **Filename: il.co.egv-...** Upload Timestamp: 2014-11-20 11:46:10 UTC Start Timestamp: 2014-11-20 11:46:11 UTC Total Duration: 1 mins Static Analysis Status: Completed Dynamic Analysis Status: Completed Action(s): Remove		

Figure B.3: Home tab APK analysis status



Summary

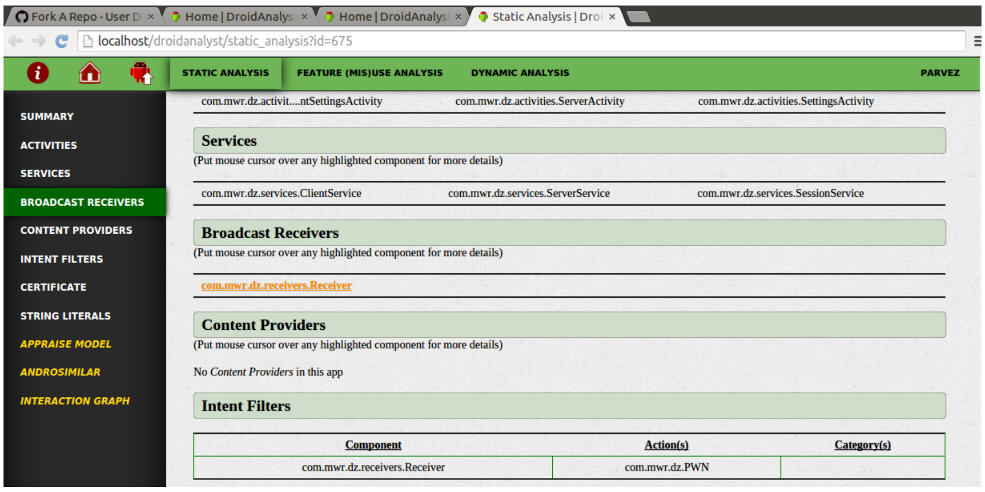
SHA-256	d4dd5bb73da3f771a2b06ea7cdb1261c633538090bc97089b731275136d67133
MD5	3ce3b0e673d7a199186e8568fcabaab2
Package Name	com.mwr.dz
Filename	agent.apk
Native code	Present
Dynamic code	Not Present
Cryptographic code	Not Present
Reflection code	Present
AndroSimilar Result	Benign

Activities
(Put mouse cursor over any highlighted component for more details)

com.mwr.dz.activities.AboutActivity	**com.mwr.dz.activities.MainActivity**	com.mwr.dz.activities.EndpointActivity
com.mwr.dz.activit....ntSettingsActivity	com.mwr.dz.activities.ServerActivity	com.mwr.dz.activities.SettingsActivity

Figure B.4: Analysis: Summary and Activities

- **Partially Completed**: Analysis is over but one or more modules did not respond properly.
- **Completed**: Analysis completed. Result can be viewed by clicking Completed link.
- ICC based Component Interaction Graph
- Permission based analysis with: ApPRaIse
- Novel malware variant signature: AndroSimilar



com.mwr.dz.activit....ntSettingsActivity com.mwr.dz.activities.ServerActivity com.mwr.dz.activities.SettingsActivity

Services

(Put mouse cursor over any highlighted component for more details)

com.mwr.dz.services.ClientService com.mwr.dz.services.ServerService com.mwr.dz.services.SessionService

Broadcast Receivers

(Put mouse cursor over any highlighted component for more details)

com.mwr.dz.receivers.Receiver

Content Providers

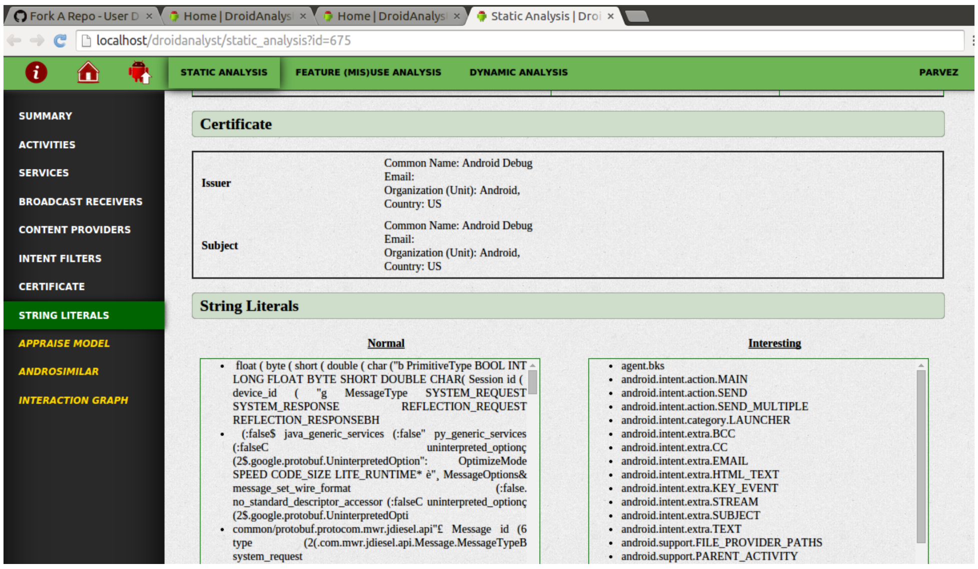
(Put mouse cursor over any highlighted component for more details)

No *Content Providers* in this app

Intent Filters

Component	Action(s)	Category(s)
com.mwr.dz.receivers.Receiver	com.mwr.dz.PWN	

Figure B.5: Analysis: Service, Broadcast Receivers, Content Providers and Intent Filters



Certificate

Issuer	Common Name: Android Debug Email: Organization (Unit): Android, Country: US
Subject	Common Name: Android Debug Email: Organization (Unit): Android, Country: US

String Literals

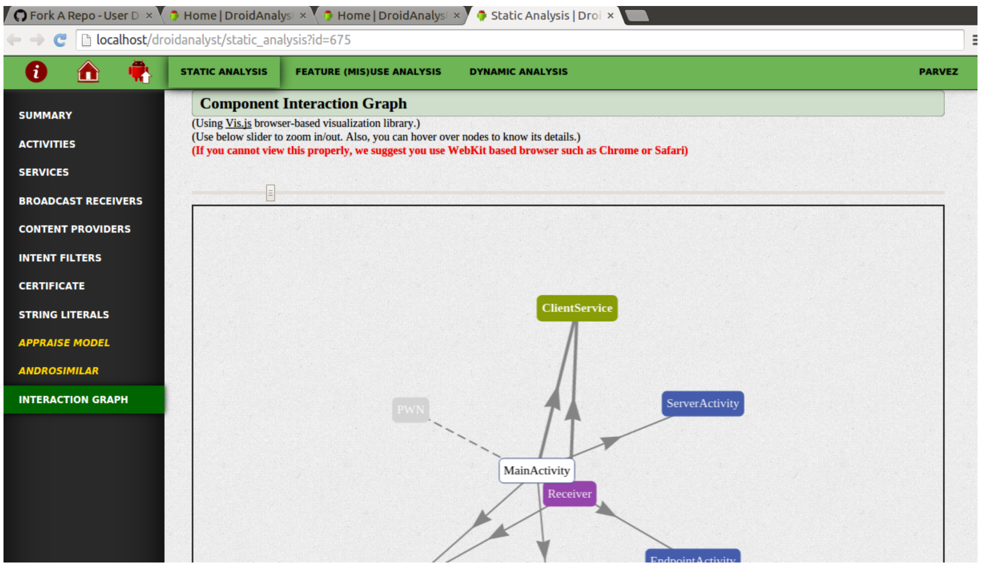
Normal

- float (byte (short (double (char ("b PrimitiveType BOOL INT LONG FLOAT BYTE SHORT DOUBLE CHAR(Session id (device_id ("g MessageType SYSTEM_REQUEST SYSTEM_RESPONSE REFLECTION_REQUEST REFLECTION_RESPONSEBH
- (:false$ java_generic_services (:false" py_generic_services (:falseC uninterpreted_optionç (2$.google.protobuf.UninterpretedOption": OptimizeMode SPEED CODE_SIZE LITE_RUNTIME* è", MessageOptions& message_set_wire_format (:false. no_standard_descriptor_accessor (:falseC uninterpreted_optionç (2$.google.protobuf.UninterpretedOpti
- common/protobuf.protocom.mwr.jdiesel.api"£ Message id (6 type (2(.com.mwr.jdiesel.api.Message.MessageTypeB system_request

Interesting

- agent.bks
- android.intent.action.MAIN
- android.intent.action.SEND
- android.intent.action.SEND_MULTIPLE
- android.intent.category.LAUNCHER
- android.intent.extra.BCC
- android.intent.extra.CC
- android.intent.extra.EMAIL
- android.intent.extra.HTML_TEXT
- android.intent.extra.KEY_EVENT
- android.intent.extra.STREAM
- android.intent.extra.SUBJECT
- android.intent.extra.TEXT
- android.support.FILE_PROVIDER_PATHS
- android.support.PARENT_ACTIVITY

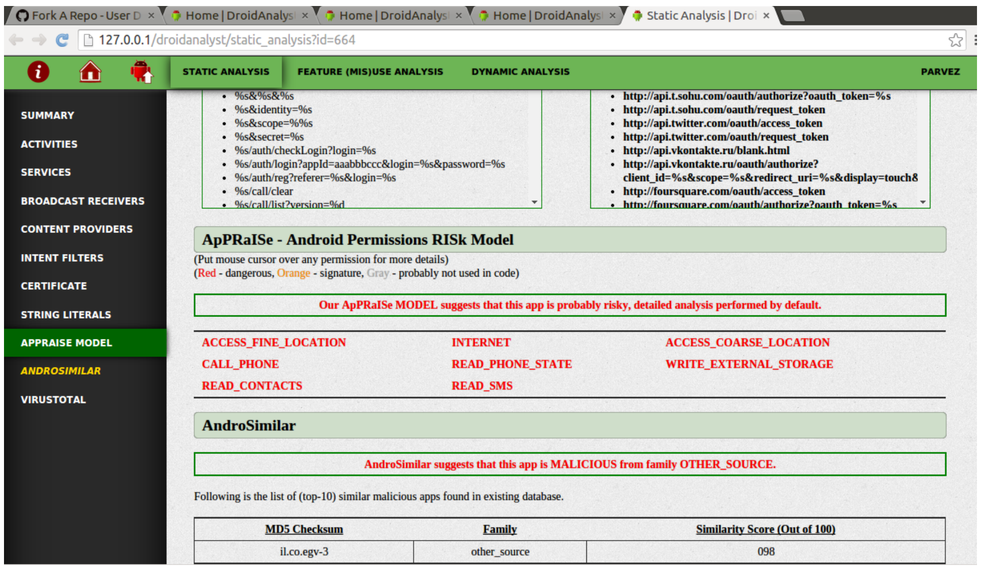
Figure B.6: Analysis: Certificate and String Literal



Component Interaction Graph

(Using Vis.js browser-based visualization library.)
(Use below slider to zoom in/out. Also, you can hover over nodes to know its details.)
(If you cannot view this properly, we suggest you use WebKit based browser such as Chrome or Safari)

Figure B.7: Analysis: Component Interaction Graph



- %s&%s&%s
- %s&identity=%s
- %s&scope=%%s
- %s&secret=%s
- %s/auth/checkLogin?login=%s
- %s/auth/login?appId=aaabbbccc&login=%s&password=%s
- %s/auth/reg?referer=%s&login=%s
- %s/call/clear
- %s/call/list?version=%d

- http://api.t.sohu.com/oauth/authorize?oauth_token=%s
- http://api.t.sohu.com/oauth/request_token
- http://api.twitter.com/oauth/access_token
- http://api.twitter.com/oauth/request_token
- http://api.vkontakte.ru/blank.html
- http://api.vkontakte.ru/oauth/authorize?client_id=%s&scope=%s&redirect_uri=%s&display=touch&
- http://foursquare.com/oauth/access_token
- http://foursquare.com/oauth/authorize?oauth_token=%s

ApPRaISe - Android Permissions RISk Model

(Put mouse cursor over any permission for more details)
(Red - dangerous, Orange - signature, Gray - probably not used in code)

Our ApPRaISe MODEL suggests that this app is probably risky, detailed analysis performed by default.

ACCESS_FINE_LOCATION	INTERNET	ACCESS_COARSE_LOCATION
CALL_PHONE	READ_PHONE_STATE	WRITE_EXTERNAL_STORAGE
READ_CONTACTS	READ_SMS	

AndroSimilar

AndroSimilar suggests that this app is MALICIOUS from family OTHER_SOURCE.

Following is the list of (top-10) similar malicious apps found in existing database.

MD5 Checksum	Family	Similarity Score (Out of 100)
il.co.egv-3	other_source	098

Figure B.8: Analysis: ApPRaIse and AndroSimilar

B.3 Capabilities

DroidAnalyst disassembles the Dalvik bytecode of an `.apk` file. The framework analyzes dangerous permissions, performs interprocedural control flow analysis, detect permission over privileges and integrates commercial antimalware with the analysis from virus total. The present functionality of DroidAnalyst is given below:

- Disassemble Android app.
- Each component (Activity, Services, Permissions, Content Providers, Broadcast Receivers, Intent filters) is highlighted if any dangerous permission usage is identified.
- Developer Certificate is extracted and presented in human readable format.
- String literals including URLs are extracted and categorized into “Normal” and “Interesting” categories.
- Predict app risk based on permissions requested and important bytecode parameters native code, reflection code, and dynamic code loading.
- Find top malicious apps that are similar to the uploaded app by extracting robust statistical features. It is done using AndroSimilar.
- Perform control flow and interprocedural data flow analysis on Dalvik Bytecode to identify Telephony, Camera, and Audio/Video misuse.
- Perform Taint Analysis to identify privacy leakage.
- Extract embedded APK files inside resources/assets of an uploaded app and automatically analyses them.
- Check uploaded APK at VirusTotal using the Google API.

At present, DroidAnalyst identifies “single malicious app”. However, the popularity of Android apps has motivated the malware authors to develop attacks based on colluding apps. There is a scope for further improvements in the analysis engine to identify and analyze the malicious app collusion.

B.4 MNIT Android Dataset

The experimental evaluation requires a substantial dataset of benign and malicious apps. We implemented a web-based APK crawler to gather the apps. 27,432 benign apps were crawled from Google Play, Anzi, HiAPK, and other Asian app markets. A total 47,342 apps were crawled from Google Play, Anzhi, and other third party Asian markets between January 2013 and September 2014; 26,469 malicious apps are downloaded from virus share, contagiominidump, and other third party markets. We also received 312 unique malware samples at DroidAnalyst based on anonymous user uploads.

We obtained malicious apps from known malware repositories Android malware genome [2], Droidbench [146] and IccRE [84]. Furthermore, the APK crawler data and user submissions were classified into MNIT Android malware database consisting 647 malware. We labelled the crawled and received malicious apps and classified them into 70 malware families. The MNIT dataset prepared in 2014-2015 is available to the researchers on request. Table B.1 lists the MNIT malware dataset.

Table B.1: MNIT Classified Malware dataset

MNIT Android Malware Dataset				
Legacy	Lotoor	Luckycat	Mania	Oldboot
51cool	AdSmS	FakeDoc	NandroBox	OpFake
Ackposts	Agent	FakeFacebook	Nyleaker	SimpleLocker
cawitt	Airpush	FakeFlashPlayer	Penetho	Skullkey
Cellspy	Antammi	FakeInst	Pincer	SmsSilence
Chuli	Antares	FakeMart	Scavir	SMSSniffer
Cosha	ArSpam	FakeRegSms	Seaweth	Steek
CounterClank	BadNews	FakeTimer	Tetus	Stiniter
DroidSheep	FakeAngry	FakeUpdates	Updtkiller	Suspicious
Flexispy	FakeAV	Fatakr	Uracto	Tascudap
Gamex	FakeBank	Fidall	Uten	Biige
Gedma	FakeDaum	FinSpy	WalkinWat	MMarketPay
Imlog	FakeDefender	Fjcon	Zeahache	
Killermob	Koler	Ksapp	Nandrobox	

B.5 List of Acronyms

ALSR Address Space Layout Randomization

ADB Android Debug Bridge

AOSP Android Open Source Project

API Application Programming Interface

ARM Advanced RISK Machines

APK Android PacKage File

AVD Android Virtual Device

AST Abstract Syntax Tree

CIG Component Interaction Graph

CTPH Context Trigger Piecewise Hashing

CDMA Code Division Multiple Access

CFG control-flow graph

DAC Discretionary Access Control

DRM Digital Rights Management

DEX Dalvik EXecutable

DVM Dalvik Virtual Machine

DNS Domain Name Resolution

DDI Dalvik Dynamic Instrumentation

DoS Denial of Service

FP False Positive

FN False Negative

FNR False Negative Rate

FPR False Positive Rate

GSM Global System for Mobile communications

GPS Global Positioning System

GUI Graphical User Interface

GID Group IDentifier

ICC Inter-Component Communication

IPC Inter-Process Communication

IDS Intrusion Detection System

IMEI International Mobile Equipment Identity

IMSI International Mobile Subscriber Identity

IoT Internet of Things

iOS iPhone Operating System

MID Mobile Internet Devices

mrMR minimum redundancy Maximum Relevance

MU Multiple Users

MAC Mandatory Access Control

NFC Near Field Communication

NOP no-operation code

NDK Native Development Kit

NX No eXecute

NCD Normalized Compression Distance

OHA Open Handset Alliance

OEM Original Equipment Manufacturer

OS Operating System

ODEX optimized dalvik executable

PC	Personal Computer
PUA	Potentially Unwanted Apps
RP	Restricted Profile
SDHash	Similarity Digest Hashing
SELinux	Security Enhanced Linux
SMS	Short Message Service
TP	True Positive
TWU	Threat Weighting Unit
UI	User Interface
UID	Unique IDentifier
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VMI	Virtual Machine Introspection
Wi-Fi	Wireless Fidelity

List of Contributions

A. Journal Publications

[J-1] **Parvez Faruki**, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, M. S. Gaur, Mauro Conti, and Raj Muttukrishnan. “Android Security: A Survey of issues, Malware penetration and Defenses”. *(IEEE) Communications Surveys and Tutorials*, Volume 17 (2), second quarter 2015, pages:998-1022.

[J-2] **Parvez Faruki**, Vijay Ganmoor, Vijay Laxmi, M. S. Gaur, and Ammar Bharmal. “AndroSimilar: Robust Signature for Variants of Android Malware”, *Journal of Information Security and Applications*, Elsevier-JISA, Available online 18 November 2014.

[J-3] **Parvez Faruki**, Hossein Fereidooni, Vijay Laxmi, M. S. Gaur, and Mauro Conti. “Android Code Obfuscation: Review of Past, Present and Future directions”, *In IEEE Transactions on Services Computing*, (IEEE TSC). **submitted**.

B. Book Chapter

[B-1] **Parvez Faruki**, Vijay Laxmi, Manoj Gaur, Shweta Bhandari, and Mauro Conti, “*DroidAnalyst: Synergic App framework for static and dynamic app analysis*”, Recent Advances in Computational Intelligence in Defense and Security, Studies in Computational Intelligence 621, DOI 10.1007/978-3-319-26450-9$_2$0$, OttawaSection, Canada.$

C. Conference Publications

[C-1] **Parvez Faruki**, Vijay Ganmoor, Vijay Laxmi, M. S. Gaur, and Ammar Bharmal, "AndroSimilar: Robust Statistical Feature Signature for Android Malware Detection". In *Proceedings of the sixth ACM conference on Security of Information and Networks*, University of Aksaray, Turkey. ACM, 2013.

[C-2] **Parvez Faruki**, Vijay Ganmoor, Vijay Laxmi, M. S. Gaur, and Ammar Bharmal, "DroidOLytics: Robust Feature Signature for Repackaged Android Apps on Official and Third Party Markets". In *Proceedings of the second IEEE conference on Advanced Computing, Networking and Security*, National Institute of Technology, Surathkal (India), 2013.

[C-3] **Parvez Faruki**, Ammar Bharmal, Vijay Laxmi, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. "Evaluation of Android anti-malware techniques against Dalvik bytecode obfuscation". In *Proceedings of the 13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, (TrustCom 2014), Beijing, China, September 24-26, 2014, pp. 414-421.

[C-4] **Parvez Faruki**, Vijay Ganmoor, Vijay Laxmi, Manoj Gaur, and Mauro Conti. "Android Platform Invariant Sandbox for Analyzing Malware and Resource Hogger apps". In *Proceedings of the 10th IEEE International Conference on Security and Privacy in Communication Networks*, (SecureComm 2014), Beijing China, 26-28 September 2014.

[C-5] **Parvez Faruki**, M.S. Gaur, Vijay Ganmoor, Ammar Bharmal and Vijay Laxmi, "Machine Learning Approach Based on Code Similarity to Detect Unknown Malicious Samples". In *IEEE Workshop on Computational Intelligence: Theories, Applications and Future Directions*, IIT Kanpur, India, pp. 208-213, July 2013. **Best paper and Best presentation award**.

[C-6] **Parvez Faruki**, Vijay Laxmi, Manoj Singh Gaur, Vinod P., "Control Flow Graph as Signature to Detect Portable Executable Malware". Poster, In *Second Security & Privacy Symposium*, (SP Symposium), February 28-March 2, 2013, IIT Kanpur. **Best Poster Award**.

Bibliography

[1] Android Security Overview. `http://source.android.com/devices/tech/security`, Online; accesed December 2013.

[2] Android Malware Genome Project. http://www.malgenomeproject.org/, Online; accessed February 2013.

[3] F-Secure: Mobile Threat Report H1 2013. http://www.f-secure.com/static/doc/labs_global/Research/Threat_Report_H1_2013.pdf, Online; accessed July 2013.

[4] F-Secure: Mobile Threat Report Q3 2013. http://www.f-secure.com/static/doc/labs_global/Research/Mobile_Threat_Report _Q3_2013.pdf, Online; accessed February 2014.

[5] ESET: Trends for 2013. http://go.eset.com/us/resources/white-papers/Trends_for_2013_preview.pdf, Online; accessed June 2013.

[6] McAfee Labs Threats Report: Third Quarter 2013. http://www.mcafee.com/uk/resources/reports/rp-quarterly-threat-q3-2013.pdf, Online; accessed January 2014.

[7] Kaspersky Security Bulletin 2013. Overall statistics for 2013. https://www.securelist.com/en/analysis/204792318/Kaspersky_Security _Bulletin_2013_Overall_statistics_for_2013, Online; accessed February 2013.

[8] Idc: Smartphone os marketshare 2015, 2014, 2013. http://www.idc.com/prodserv/smartphone-os-market-share.jsp, June 2015.

[9] Gartner Inc. Android Smartphone Sales Report, 2013. `http://www.gartner.com/newsroom/id/2665715`, Online; accessed March 2014.

[10] Carlos A. Castillo. Android Malware Past, Present, and Future. Technical report, Mobile Working Security Group McAfee, 2012.

[11] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng 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, pages 317–326, New York, NY, USA, 2012. ACM.

[12] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*, SP '07, pages 231–245, Washington, DC, USA, 2007. IEEE Computer Society.

[13] Thorsten Holz, Markus Engelberth, and Felix Freiling. Learning more about the underground economy: A case-study of keyloggers and dropzones. In *Proceedings of the 14th European Conference on Research in Computer Security*, ESORICS'09, pages 1–18, Berlin, Heidelberg, 2009. Springer-Verlag.

[14] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Evaluation of android anti-malware techniques against dalvik bytecode obfuscation. In *13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2014, Beijing, China, September 24-26, 2014*, pages 414–421. IEEE, 2014.

[15] Michael Becher, Felix C. Freiling, Johannes Hoffmann, Thorsten Holz, Sebastian Uellenbeck, and Christopher Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 96–111, Washington, DC, USA, 2011. IEEE Computer Society.

[16] Lookout Inc. Current World of Mobile Threats. Technical report, Lookout Mobile Security, 2013.

[17] Michael Spreitzenbarth. *Dissecting the Droid: Forensic Analysis of Android and its malicious Applications.* PhD thesis, 2013.

[18] AppBrain. Number of applications available on Google Play. `http://www.appbrain.com/stats/number-of-android-apps`, Online; accessed October 2014.

[19] 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/, Online; accessed February 2012.

[20] Android and security: Official mobile google blog. http://googlemobile.blogspot.in/2012/02/android-and-security.html, Online; accessed October 2013.

[21] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 627–638, New York, NY, USA, 2011. ACM.

[22] Baidu. http://as.baidu.com/, Online; accessed March 2014.

[23] AppChina. http://www.appchina.com/, Online; accessed March 2014.

[24] Earlence Fernandes, Bruno Crispo, and Mauro Conti. Fm 99.9, radio virus: Exploiting fm radio broadcasts for malware deployment. *IEEE Transactions on Information Forensics and Security*, 8(6):1027–1037, 2013.

[25] Rafael Fedler, Julian Schütte, and Marcel Kulicke. On the Effectiveness of Malware Protection on Android. Technical report, Technical report, Fraunhofer AISEC, Berlin, 2013.

[26] Chris Jarabek, David Barrera, and John Aycock. ThinAV: Truly lightweight Mobile Cloud-based Anti-malware. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 209–218. ACM, 2012.

[27] Kaspersky Internet Security for Android. http://www.kaspersky.com/android-security, Online; accessed July 2013.

[28] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian 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, pages 281–294, New York, NY, USA, 2012. ACM.

[29] Zhou Yajin and Jiang Xuxian. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, Oakland 2012. IEEE, 2012.

[30] Sebastian Schrittwieser, Stefan Katzenbeisser, Peter Kieseberg, Markus Huber, Manuel Leithner, Martin Mulazzani, and Edgar Weippl. Covert computation hiding code in code through compile-time obfuscation. *Computers & Security*, 42(0):13 – 26, 2014.

[31] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime data in android applications for identifying malware and enhancing code analysis. Technical Report TUD-CS-2015-0031, EC SPRIDE, February 2015.

[32] Mcafee labs threats report: May 2015. www.mcafee.com/in/resources/reports/rp-quarterly-threat-q1-2015.pdf, Online; accessed August 2015.

[33] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding android security. *IEEE Security and Privacy*, 7(1):50–57, January 2009.

[34] RageAgainstTheCage. https://github.com/bibanon/android-development-codex/blob/master/General/Rooting/rageagainstthecage.md, Online; accessed February 2013.

[35] GingerBreak. http://forum.xda-developers.com/showthread.php?t=1044765, Online; accessed February 2013.

[36] z4Root. https://github.com/bibanon/android-development-codex/blob/master/General/Rooting/z4root.md, Online; accessed February 2013.

[37] Android Security Analysis Challenge: Tampering Dalvik Bytecode During Runtime. http://bluebox.com/labs/android-security-challenge/, Online; accessed February 2013.

[38] Lookout Inc. State of Mobile Security 2012. Technical report, Lookout Mobile Security, 2012.

[39] Charles Lever, Manos Antonakakis, Brad Reaves, Patrick Traynor, and Wenke Lee. The Core of the Matter: Analyzing Malicious Traffic in Cellular Carriers. In *Proc. NDSS*, volume 13, pages 1–16.

[40] Hien Thi Thu Truong, Eemil Lagerspetz, Petteri Nurmi, Adam J Oliner, Sasu Tarkoma, N Asokan, and Sourav Bhattacharya. The Company You

Keep: Mobile Malware Infection Rates and Inexpensive Risk Indicators. *arXiv preprint arXiv:1312.3245*, 2013.

[41] Carat: Collaborative Energy Diagnosis. http://carat.cs.berkeley.edu/, Online; accesed December 2013.

[42] APKTool. Reverse Engineering with Apktool. `https://code.google.com/android/apk-tool`, Online; accessed March, 2014.

[43] Android Inc. Class to Dex Conversion with Dx. `http://developer.android.com/tools/help/index.html`, Online; accessed March 2013.

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

[45] Android/NotCompatible Looks Like Piece of PC Botnet. http://blogs.mcafee.com/mcafee-labs/androidnotcompatible-looks-like-piece-of-pc-botnet, Online; accesed December 2013.

[46] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: Evaluating Android anti-malware against Transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[47] Min Zheng, Patrick P. C. Lee, and John C. S. Lui. ADAM: An Automatic and Extensible Platform to Stress Test Android Anti-virus Systems. In *DIMVA*, pages 82–101, 2012.

[48] ProGuard: Free java class file shrinker, optimizer and obfuscator. http://proguard.sourceforge.net/, Online; accessed March 2014.

[49] DexGuard. http://www.saikoa.com/dexhuang2014asdroidguard, Online; accessed February 2013.

[50] P. Faruki, A. Bharmal, V. Laxmi, M. Gaur, M. Conti, and R. Muttukrishnan. Evaluation of android anti malware techniques against dalvik bytecode obfuscation, to appear. In *proceedings of the 13th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom'14), to appear,Beijing China, 26-28 Sept. 2014.*, September 2014.

[51] Dalvik Bytecode Obfuscation on Android. https://dexlabs.org/blog/bytecode-obfuscation, Online; accessed February 2013.

[52] BlackHat. Reverse Engineering with Androguard. `https://code.google.com/androguard`, Online; accessed March 2014.

[53] Strazzare. ANDROID HACKER PROTECTION LEVEL. `https://www.defcon.org/images/defcon-22/\\dc-22-presentations/Strazzere-Sawyer/DEFCON-22-Strazzere-and-Sawyer-Android-Hacker-Protection-Level\\-UPDATED.pdf`, Online; accessed May 2014.

[54] Patrick Schulz. Code protection in android. `https://net.cs.uni-bonn.de/fileadmin/user_upload/plohmann/2012-Schulz-Code_Protection_in_Android.pdf`, Online; accessed December 2012.

[55] Strazzere. Dex education: Practicing safe dex. `http://www.strazzere.com/papers/DexEducation-PracticingSafeDex.pdf`, Online; accessed December 2013.

[56] Axelle Apvrille. Angecryption: Hide android applications in images. `https://www.blackhat.com/docs/eu-14/materials/eu-14-Apvrille-Hide-Android-Applications-In-Images-wp.pdf`, Online; accessed November 2014.

[57] Wu Zhou, Yajin Zhou, and Xuxian Jiang. Hey, You Get Off my Market: Detecting Malicious apps in Official and Third party Android Markets. In *Annual Network and Distributed Security Symposium*, New York, NY, USA, 2012. NDSS.

[58] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "andromaly": a behavioral malware detection framework for android devices. *J. Intell. Inf. Syst.*, 38(1):161–190, 2012.

[59] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 576–587, New York, NY, USA, 2014. ACM.

[60] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, MobiSys '11, pages 239–252, New York, NY, USA, 2011. ACM.

[61] Adam P Fuchs, Avik Chaudhuri, and Jeffrey S Foster. SCanDroid: Automated security certification of Android applications. *Manuscript, Univ. of Maryland, http://www. cs. umd. edu/˜ avik/projects/scandroidascaa*, 2009.

[62] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. CHEX: Statically vetting Android apps for Component hijacking vulnerabilities. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM Conference on Computer and Communications Security*, pages 229–240. ACM, 2012.

[63] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android Permissions: a perspective combining risks and benefits. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, pages 13–22. ACM, 2012.

[64] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil 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, pages 73–84, New York, NY, USA, 2010. ACM.

[65] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. PUMA: Permission usage to detect malware in android. In *International Joint Conference CISIS12-ICEUTE´ 12-SOCO´ 12 Special Sessions*, pages 289–298. Springer, 2013.

[66] Chun-Ying Huang, Yi-Ting Tsai, and Chung-Han Hsu. Performance Evaluation on Permission-Based Detection for Android Malware. In *Advances in Intelligent Systems and Applications-Volume 2*, pages 111–120. Springer, 2013.

[67] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

[68] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Network and Distributed System Security Symposium (NDSS)*, February 2012.

[69] Jinyung Kim, Yongho Yoon, Kwangkeun Yi, Junbum Shin, and SWRD 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.

[70] Henrik Søndberg Karlsen, Erik Ramsgaard Wognsen, Mads Chr Olesen, and René Rydhof 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.

[71] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In Tanveer Zia, Albert Y. Zomaya, Vijay Varadharajan, and Zhuoqing Morley Mao, editors, *SecureComm*, volume 127 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 86–103. Springer, 2013.

[72] Min Zheng, Mingshen Sun, and John C. S. Lui. DroidAnalytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware. *CoRR*, abs/1302.7212, 2013.

[73] JD-GUI. Android Decompiling with JD-GUI. `http://java.decompiler.free.fr/?q=jdgui`, Online; accessed March 2014.

[74] JAD. JAD Java Decompiler. `http://varaneckas.com/jad/`, Online; accessed March 2014.

[75] Hanpeter van Vliet. Mocha, The Java Decompiler. `http://www.brouhaha.com/~eric/software/mocha/`, Online; accessed March 2014.

[76] SOOT. Soot: a java optimization framework. `http://www.sable.mcgill.ca/soot/`, Online; accessed March 2014.

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

[78] HP Inc. Fortify static code analyzer. http://www8.hp.com/us/en/software-solutions/software.html?compURI=1338812, Online; accessed March 2014.

[79] Enk William, Octeau Damien, McDaniel Patrick, and Chaudhari Swarat. A Study of Android Application Security. In *USENIX Security '11*, San Francisco, ca, 2011. USENIX.

[80] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting Android applications to Java bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 6. ACM, 2012.

[81] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin 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*, pages 27–38. ACM, 2012.

[82] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: Automatically detecting potential privacy leaks in Android applications on a large scale. In *Trust and Trustworthy Computing*, pages 291–307. Springer, 2012.

[83] Dex2Jar. Android Decompiling with Dex2jar. `http://code.google.com/p/dex2jar/`, Online; accessed May 2013.

[84] Li Li, Alexandre Bartel, Jacques Klein, Yves Le Traon, Steven Arzt, Rasthofer Siegfried, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis. Technical Report 978-2-87971-129-4_TR-SNT-2014-9, April 2014.

[85] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android with epicc: An essential step towards holistic security analysis. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 543–558, Berkeley, CA, USA, 2013. USENIX Association.

[86] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-

aware taint analysis for android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 259–269, New York, NY, USA, 2014. ACM.

[87] UI/Application Exercise Monkey. http://developer.android.com/tools/help/monkey.html, Online; accessed February 2013 2013.

[88] Jon Oberhide. Dissecting The Android Bouncer. `http://jon.oberheide.org/blog/2012/06/21/dissecting-the-android-bouncer/?`, Online; accessed June 2013.

[89] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 447–458, New York, NY, USA, 2014. ACM.

[90] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A System call-centric analysis and stimulation technique to automatically reconstruct Android Malware behaviors. *EUROSEC, Prague, Czech Republic*, 2013.

[91] Dimitrios Damopoulos, Georgios Kambourakis, and Georgios Portokalidis. The best of both worlds: A framework for the synergistic operation of host and cloud anomaly-based ids for smartphones. In *Proceedings of the Seventh European Workshop on System Security*, EuroSec '14, pages 6:1–6:6, New York, NY, USA, 2014. ACM.

[92] Enk William, Gilbert Peter, Chun Byunggon, and Cox Landon. TaintDroid : An Information Flow Tracking System for Realtime Privacy monitoring on Smartphones. In *USENIX Symposium on Operating Systems Design and Implementation*. USENIX, 2011.

[93] Iker Burguera, Urko Zurutuza, and Simin 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*, SPSM '11, pages 15–26, New York, NY, USA, 2011. ACM.

[94] Karim O. Elish, Danfeng (daphne Yao), and Barbara G. Ryder. User-centric dependence analysis for identifying malicious mobile apps, 2012.

[95] Jianjun Huang, Xiangyu Zhang, Lin Tan, Peng Wang, and Bin Liang. Asdroid: detecting stealthy behaviors in android applications by user interface and program behavior contradiction. In *ICSE*, pages 1036–1046, 2014.

[96] Karim O. Elish, Xiaokui Shu, Danfeng (Daphne) Yao, Barbara G. Ryder, and Xuxian Jiang. Profiling user-trigger dependence for android malware detection. *Computers & Security*, 49:255 – 273, 2015.

[97] Fatemeh Azmandian, Micha Moffie, Malak Alshawabkeh, Jennifer Dy, Javed Aslam, and David Kaeli. Virtual machine monitor-based lightweight intrusion detection. *SIGOPS Oper. Syst. Rev.*, 45(2):38–53, July 2011.

[98] Xuxian Jiang, Xinyuan Wang, and Dongyan Xu. Stealthy malware detection through vmm-based ”out-of-the-box” semantic view reconstruction. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS ’07, pages 128–138, New York, NY, USA, 2007. ACM.

[99] Brendan Dolan-Gavitt, Tim Leek, Michael Zhivich, Jonathon Giffin, and Wenke Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP ’11, pages 297–312, Washington, DC, USA, 2011. IEEE Computer Society.

[100] Lok Kwong Yan and Heng Yin. Droidscope: Seamlessly reconstructing the OS and Dalvik Semantic views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[101] Parvez Faruki, Vijay Ganmoor, Vijay Laxmi, M. S. Gaur, and Ammar Bharmal. AndroSimilar: Robust Statistical Feature Signature for Android Malware Detection. In *Proceedings of the 6th International Conference on Security of Information and Networks*, SIN ’13, pages 152–159, New York, NY, USA, 2013. ACM.

[102] Andrubis: A tool for analyzing unknown android applications. http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2/, Online; accesed December 2012.

[103] APKInspector. https://github.com/honeynet/apkinspector/, Online; accesed December 2013.

[104] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy Enforcement for Android applications. In *Proceedings of the 21st USENIX conference on Security symposium*, pages 27–27. USENIX Association, 2012.

[105] CopperDroid. http://copperdroid.isg.rhul.ac.uk/copperdroid/index.php, Online: accessed February 2015.

[106] Anthony Desnos and Patrik Lantz. Droidbox: An android application sandbox for dynamic analysis. https://code.google.com/p/droidbox/, 2011.

[107] Drozer - A Comprehensive Security and Attack Framework for Android. https://www.mwrinfosecurity.com/products/drozer/, Online; accessed February 2013.

[108] JEB Decompiler. http://www.android-decompiler.com/, Online; accessed February 2013.

[109] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: Versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 347–356, New York, NY, USA, 2010. ACM.

[110] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe exposure analysis of mobile in-app advertisements. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 101–112, New York, NY, USA, 2012. ACM.

[111] Annamalai Narayanan, Lihui Chen, and Chee Keong Chan. Addetect: Automated detection of android ad libraries using semantic analysis. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), Singapore, April 21-24, 2014*, pages 1–6. IEEE, 2014.

[112] Hao Dong, Jieqi Kang, James Schafer, and Aura Ganz. Android-based visual tag detection for visually impaired users: System design and testing. *Int. J. E-Health Med. Commun.*, 5(1):63–80, January 2014.

[113] Dendroid malware can take over your camera, record audio, and sneak into google play. https://blog.lookout.com/blog/2014/03/06/dendroid/, Online; accessed April 2014.

[114] Parvez Faruki, Vijay Ganmoor, Laxmi Vijay, Manoj Gaur, and Mauro Conti. Android Platform Invariant Sandbox for Analyzing Malware and Resource Hogger apps. In *Proceedings of the 10th IEEE International Conference on Security and Privacy in Communication Networks (SecureComm 2014), Beijing China, 26-28 September 2014.* Securecomm, September 2014.

[115] Guillermo Suarez-Tangil, Mauro Conti, Juan E. Tapiador, and Pedro Peris-Lopez. Detecting targeted smartphone malware with behavior-triggering stochastic models. In *In Proceedings of the European Symposium on Research in Computer Security, to appear*, ESORICS 2014. ESORICS, 2014.

[116] Paul Ratazzi, Yousra Aafer, Amit Ahlawat, Hao Hao, Yifei Wang, and Wenliang Du. A systematic security evaluation of android's multi-user framework. In *Proceedings of the IEEE Mobile Security Technologies Workshop, MoST*. IEEE, 5 2014.

[117] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[118] Martina Lindorfer. Andrubis - 1,000,000 apps later: A view on current android malware behaviors. *http://www.seclab.tuwien.ac.at/papers/andrubis_badgers14.pdf*, Online; accesed December 2014.

[119] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[120] Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis: Android Malware Under The Magnifying Glass. Technical Report TR-ISECLAB-0414-001, Vienna University of Technology, 2014.

[121] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Çamtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *MALWARE*, pages 55–62, 2010.

[122] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 93–104, New York, NY, USA, 2012. ACM.

[123] Sebastian Neuner, Victor Van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar R. Weippl. Enter sandbox: Android sandbox comparison. In *Proceedings of the IEEE Mobile Security Technologies Workshop, MoST*. IEEE, 5 2014.

[124] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1808–1815, New York, NY, USA, 2013. ACM.

[125] Parvez Faruki. Droidanalyst: Apk analysis engine. `https://www.droidanalyst.org`, 2014.

[126] Google Play. Official Android Market. `https://market.android.com`, Online; accessed June 2013.

[127] Contagiodump. Contagio Malware Dump. `http://contagiodump.blogspot.in/`, Online; accessed March 2013.

[128] virusshare malware repository. Virus Share Malware Repository. `http://www.virusshare.com/`, Online; accessed March 2014.

[129] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. Pscout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, New York, NY, USA, 2012. ACM.

[130] Mumayi.com. Third-party app-store, china. `http://www.mumayi.com/`, Online; accessed January 2014.

[131] gfan.com. Third-party app-store, china. `http://www.gfan.com/`, Online; accessed January 2014.

[132] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[133] Hanchuan Peng, Fuhui Long, and Chris Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 27(8):1226–1238, 2005.

[134] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman M. Sadeh, and David Wetherall. A conundrum of permissions: Installing applications on an android smartphone. In Jim Blythe, Sven Dietrich, and L. Jean Camp, editors, *Financial Cryptography Workshops*, volume 7398 of *Lecture Notes in Computer Science*, pages 68–79. Springer, 2012.

[135] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: user attention, comprehension, and behavior. In Lorrie Faith Cranor, editor, *SOUPS*, page 3. ACM, 2012.

[136] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2012.

[137] Vassil Roussev. Data Fingerprinting with Similarity Hashes. *Advances in Digital Forensics.*, 2011.

[138] Vassil Roussev. Building a better similarity trap with statistically improbable features. In *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*, pages 1–10. IEEE, 2009.

[139] Vassil Roussev. An Evaluation of Forensic Similarity Hashes. *Digit. Investig.*, 8:S34–S41, August 2011.

[140] Android Third party app store. Third-party app-store, china. `http://www.android.d.cn/`, Online; accessed January 2014.

[141] Hiapk.com. Third-party app-store, china. `http://www.hiapk.com/`, Online; accessed January 2014.

[142] Jesse Kornblum. Identifying Identical files using Context Triggered Piecewise Hashing. *Digital Investigation.*, 3:91–97, Sept. 2006.

[143] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications, 2012.

[144] Ruchna Nigam Axelle Apvrille. Obfuscation in android malware, and how to fight back. `http://www.strazzere.com/papers/DexEducation-PracticingSafeDex.pdf`, Online; accessed September 2014.

[145] Fakeinstaller leads the attack on android phones. `https://blogs.mcafee.com/mcafee-labs/fakeinstaller-leads-the-attack-on-android-phones`, Online; 2012.

[146] Droidbench-benchmarks — secure software engineering. http://sseblog.ec-spride.de/tools/droidbench/, Online; accessed February 2015.

[147] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[148] VirusTotal. https://www.virustotal.com/, Online; accessed February 2013.

[149] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 291–307, Berlin, Heidelberg, 2012. Springer-Verlag.

[150] Brandon Amos, Hamilton A. Turner, and Jules White. Applying machine learning classifiers to dynamic android malware detection at scale. In Roberto Saracco, Khaled Ben Letaief, Mario Gerla, Sergio Palazzo, and Luigi Atzori, editors, *IWCMC*, pages 1666–1671. IEEE, 2013.

[151] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*. The Internet Society, 2014.

[152] Backdoor.AndroidOS.Obad.a. http://contagiominidump.blogspot.in/2013/06/backdoorandroidosobada.html, Online; accesed December 2013.

[153] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, Scalable Detection of "Piggybacked" Mobile Applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 185–196. ACM, 2013.

[154] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting Android Applications to Java Bytecode. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 6:1–6:11, New York, NY, USA, 2012. ACM.

[155] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 329–334, New York, NY, USA, 2013. ACM.

[156] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.* Addison-Wesley Professional, 1st edition, 2009.

[157] Aleksandrina Kovacheva. Efficient code obfuscation for android. In Borworn Papasratorn, Nipon Charoenkitkarn, Vajirasak Vanijja, and Vithida Chongsuphajaisiddhi, editors, *IAIT*, volume 409 of *Communications in Computer and Information Science*, pages 104–119. Springer, 2013.

[158] Mark Stamp and Wing Wong. Hunting for metamorphic engines. *Journal in Computer Virology*, 2(3):211–229, December 2006.

[159] Michael Batchelder and Laurie J. Hendren. Obfuscating java: The most pain for the least gain. In Shriram Krishnamurthi and Martin Odersky, editors, *CC*, volume 4420 of *Lecture Notes in Computer Science*, pages 96–110. Springer, 2007.

[160] Sable Mcgill. Java obfuscation techniques. `www.sable.mcgill.ca/JBCO/examples.html`, Online; accessed June 2015.

[161] BakSmali. Reverse Engineering with Smali/Baksmali. `https://code.google.com/smali`, Online; accessed March 2013.

[162] Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu. A Framework for Evaluating Mobile App Repackaging Detection Algorithms. In *TRUST*, pages 169–186, 2013.

[163] AV-Test. AV-Test, The Independent IT-Security Institute. `http://www.av-test.org/en/home/`, Online; accessed March 2014.

[164] Google. Android tools: Adb, emulator, avd manager, android, mksdcard, monkey, logcat. `https://tools.android.com`, 2009.

[165] Vaibhav Rastogi, Yan Chen, and William Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 209–220, New York, NY, USA, 2013. ACM.

[166] TCPDump. Tcpdump public repository. `http://www.tcpdump.org/#latest-release`, Online; accessed October 2014.

[167] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the 6th European Workshop on System Security (EUROSEC 2013)*, Prague, Czech Republic, 2013.

[168] Collin Mulliner. Dalvik dynamic instrumentation, October 2013.

[169] CVE. http://cve.mitre.org/, Online; accessed February 2013.

[170] Goujon Andre and Pablo Ramos. BOXER SMS Trojan. Technical report, ESET Latin American Lab, 2013.

[171] Android Trickery. http://c-skills.blogspot.com/2010/07/android-trickery.html, Online; accessed February 2013.

[172] Zimperlich Sources. http://c-skills.blogspot.in/2011/02/zimperlich-sources.html, Online; accessed February 2013.

[173] zergrush. http://forum.xda-developers.com/showthread.php?t=1296916, Online; accessed February 2013.

[174] Security Enhancements in Android 4.3. http://source.android.com/devices/tech/security/ enhancements43.html, Online; accesed December 2013.

[175] Security Enhancements in Android 4.2. http://source.android.com/devices/tech/security/ enhancements42.html, Online; accesed December 2013.

[176] Validating Security-Enhanced Linux in Android. http://source.android.com/devices/tech/security/se-linux.html, Online; accesed December 2013.

[177] Mauro Conti, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich. CRêPe: A system for enforcing fine-grained context-related policies on android. *Information Forensics and Security, IEEE Transactions on*, 7(5):1426–1438, 2012.

[178] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In Dengguo Feng, David A. Basin, and Peng Liu, editors, *ASIACCS*, pages 328–332. ACM, 2010.

[179] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. Xmandroid: A new android evolution to mitigate privilege escalation attacks. *Technische Universität Darmstadt, Technical Report TR-2011-04*, 2011.

[180] Machigar Ongtang, Stephen E. McLaughlin, William Enck, and Patrick Drew McDaniel. Semantically rich application-centric security in android. In *ACSAC*, pages 340–349. IEEE Computer Society, 2009.

[181] DARE: Dalvik Retargeting. http://siis.cse.psu.edu/dare/, Online; accessed February 2013.

[182] Dedexer. http://dedexer.sourceforge.net/, Online; accessed February 2013.

[183] Similarities for Fun & Profit. Online; accessed February 2013.

[184] ded: Decompiling Android Applications. http://siis.cse.psu.edu/ded/, Online; accessed February 2013.