Università degli Studi di Cagliari

DOTTORATO DI RICERCA

INGEGNERIA ELETTRONICA ED INFORMATICA

Modeling Neglected Functions of Android Applications to Effectively Detect Malware

Mansour Ahmadi

Settore scientifico disciplinare di afferenza

ING-INF/05: Sistemi di elaborazione delle informazioni

Advisor: Prof. **Giorgio Giacinto**

Ph.D. Coordinator: Prof. **Fabio Roli**

XXIX Cycle

March 2017

Ph.D. in Electronic and Computer Engineering

Dept. of Electrical and Electronic Engineering

University of Cagliari

Modeling Neglected Functions of Android Applications to Effectively Detect Malware

Mansour Ahmadi

Advisor: Prof. **Giorgio Giacinto**

Ph.D. Coordinator: Prof. **Fabio Roli**

XXIX Cycle
March 2017

Dedicated to my soulmate wife Farideh and my parents,

who supported and encouraged me to accomplish this work.

Abstract

With more than two million applications, Android marketplaces require automatic and scalable methods to efficiently vet apps for the absence of malicious threats. On the other hand, Modern malware is designed by obfuscation characteristics, which causes an enormous growth in the number of malware samples. Classification of this huge number of malware samples on the basis of their behaviors is essential for the computer security community. Although there are a quite good number classification techniques, it is usual that sometimes researchers neglect modeling some parts of applications that might be misused by adversaries. In this thesis, we aim to show how we can improve the accuracy of malware classifiers by considering some neglected functions of Android applications. To this end, first, we do a comprehensive survey on Android security issues with a focus on application analysis. Then, we modeled three important functions of Android applications such as HTTP communication channel,

GCM communication channel, and code hiding techniques (e.g., dynamic code loading) to outperform the existing classification techniques. We prove our claim by performing experiments on a large set of Android applications and represent the power of wisely engineered features for having an effective learning-based malware classification system.

Acknowledgment

Firstly, I would like to express my honest gratitude to my advisor Prof. Giorgio Giacinto for his continuous support, motivation, and knowledge. His guidance helped me in all the time of my Ph.D. study. Moreover, He let me to freely decide and explore open issues in computer security that helped me a lot to expand my mindset about the problems.

"I was so so lucky to be supervised by Giorgio and I could not have imagined having a better mentor for my Ph.D."

Besides my advisor, I would like to thank Prof. Lorenzo Cavallaro and Prof. Johannes Kinder for their insightful comments and encouragement during my visiting period at Royal Holloway, University of London. They provided me access to their laboratory and research facilities which helped me a lot to conduct a part of this research.

My special thanks also goes to my colleagues, Battista Biggio, Santanu Kumar Dash, Guillermo Suarez-Tangil for the sleepless nights we were working together before deadlines, *and for all the fun we have had in the last three years. I would also like to thank all of my labmates in PRA Lab and S^2Lab and my colleagues such as Davide Ariu, Davide Maiorca, Igino Corona, Bahram Lavi, Simone Porru, Ambra Demontis, Paolo Russu, Matteo Mauri, Luca Piras, Carla Piras, and Roberto Jordaney.*

Last but not the least, I would like to thank whole of my family members (my wife, parents, brothers, sisters, brother in law, sister in law, and my wife's family) and my friends (true friends around me are not a few so it is a very long list if I want to name them) for supporting me spiritually throughout my Ph.D. and my life in general.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Advent of Mobile Devices

Nowadays, mobile devices are ubiquitous tools for everyday life. People use the same device in different places and for different tasks, such as at work, and during physical training, for leisure, and for issuing bank transaction. They login to different accounts offering different services, and store on the device valuable information such as their identity details, credit card credentials, health information [220]. The commonest mobile devices are smartphones and tablets whose main difference is the screen size, that affects the quantity of the information that can be displayed on the screen at once.[1] Mobile devices run operating systems that have been designed specifically for them, the most popular being Android, iOS, Windows phone and Blackberry OS. Among them, Android dominate the global smartphone market, with nearly 90% of the market share in the second quarter of 2016 [152]. Net Apps' data [188] shows that Android usage share, 68.54%, overtook iOS, with a share of 25.78%, in October of 2016. Many leading device manufacturers like HTC, Sony, Samsung, LG, Motorola rely on the Android operation system for its open source nature supported by the Open Handset Alliance

[1] http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/

whose members comprises mobile operators, handset manufacturers, semiconductor companies, software companies, and commercialization companies, with a strong role played by Google.

The architecture of the Android OS differs significantly from typical desktop OSes in terms of the design and execution of Apps. While desktop users can in principle install *any* application without any constraint, this is not possible in Android because of the permission model. In Android, each App has its own user account or user identification (ID) with specified permissions and privileges, or group IDs, which is inherited from the classic system-wide discretionary access control (DAC) employed by Linux. For example, when an App is granted a permission (e.g., to use the built-in camera), then the App is assigned to the corresponding camera group in Linux. No App by default has permissions to perform any operation, but all Apps must explicitly request permissions that must be granted by the user. Before Android version 6.0, in order for an application to be correctly installed and then executed, the user must accept all the permissions requested by the developer of the App, otherwise the installation process is canceled. Since Android version 6.0, the user can grant selective permissions to the App. The permissions are divided into fine-grained permissions, such as SMS (e.g., send, receive, read), and coarse-grained permissions, such as Internet. The mechanism of permissions is designed to provide increased security with respect to the access to data stored in the device, or acquired by the sensors, but a bad management of permissions can allow Apps to share data without the awareness of the users. In fact, Android isolates each App in a sandbox, so that an App can't access directly data managed by other Apps, and the permission model can provide for mechanisms enabling inter communication among Apps.

1.2 Security Issues of Mobile Devices

Although fame brings profit, it also attracts cybercriminals, that leverage on unseen vulnerabilities that can be exploited in particular circumstances. The first Android vulnerability was reported by Common Vulnerabilities and Exposures (CVE) [86] in 2009 and since then, 630 vulnerabilities have been reported by CVE. Although Android employs advanced techniques to protect Apps such as memory protection, and App's assets protection, the Android security architecture is still vulnerable to privacy violations, root exploits, confused deputy attacks, and collusion attacks. Moreover, malicious Apps is another major concern for security community. In addition to malicious Apps, threats also arise form vulnerabilities in benign Apps, that can open a path to miscreants. Whereas the architecture of Android is designed to enforce app protection, secure coding best practices still plays an important role in preventing attacks. Inter-Component Communication (ICC), Graphical User Interface (GUI), information leakage, and weak permissions management are just some of the issues that may affect the security of the user. Finally, security issues are also related to the portability of the device, that impose constraints on the resources of the devices compared to desktop computers (e.g., screen size, memory capacity, etc.), while providing additional functionalities (e.g., localization services), but makes them more likely to be easily lost or stolen.

Android Malware

The majority of the security issues affecting Android systems can be attributed to third party Apps rather than to the Android OS itself. Based on F-secure reports on mobile threats [115], researchers found 277 new malware families, among which 275 specifically target Android devices. As is shown in Figure 1.1, McAfee[2] reported the significant growth of mobile malware in the wild. This huge amount of mobile malware needs automated techniques to be analyzed, detected, and classified. Modern malware is designed with mutation characteristics, namely polymorphism and metamorphism, which causes an enormous growth in the number of variants of malware samples. Detection/Categorization of malware samples on the basis of their behaviors is essential for the computer security community, because they receive huge number of malware everyday, and the

[2] http://www.mcafee.com/us/resources/reports/rp-mobile-threat-report-2016.pdf

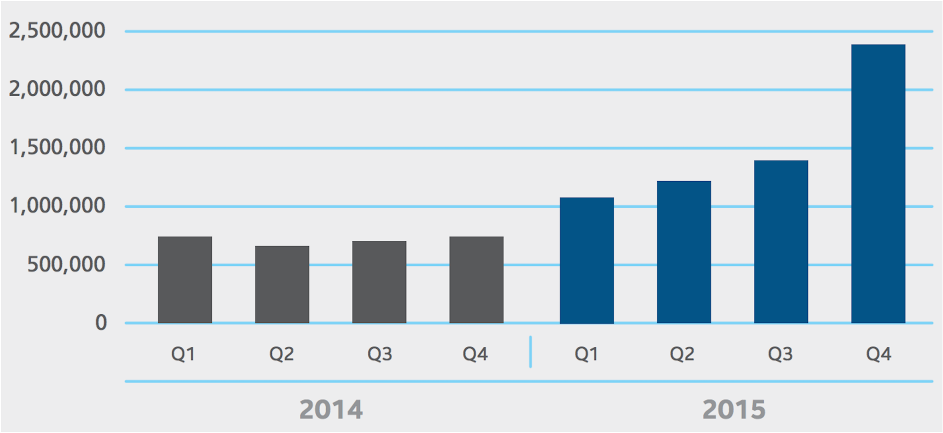


Figure 1.1: New Mobile Malware Threat Statistics

signature extraction process is usually based on malicious parts characterizing malware families.

1.3 Machine Learning for Malware Classification

As I mentioned in section 1.2, automatic classification of applications is desired for security community, and one of the tools that can facilitate the task for the community is machine learning.

Desktop Malware Classification

Machine learning has been successfully applied for desktop malware classification. The analysis of malicious programs is usually carried out by static techniques [185, 225, 277] and dynamic techniques [38, 221, 260, 265]. Analyzers extract various characteristics from the programs' syntax and semantic such as operation codes [226] and function call graph [151] from the disassembled code, and byte code n-grams [28, 246] from the hex code, or different structural characteristics from the PE header, such as dependencies between APIs [277] and DLLs [185]. Some other works [234] also explored the analysis of metadata such as the number of bitmaps, the size of import and export address tables besides the PE header's content. The aforementioned content-based detection systems,

like those considering bytecode n-grams, APIs, and assembly instructions, are inherently susceptible to false detection due to the fact of polymorphism and metamorphism. In addition, these techniques are not appropriated in the case of malware samples that does not contain any APIs, and also contains a few assembly instructions because of packing. For malware classification, we proposed a learning-based system which uses different malware characteristics to effectively assign malware samples to their corresponding families without doing any deobfuscation and unpacking process [39]. Although unpacking may lead to the extraction of more valuable features if the packers are known, unpacking is a costly task, and dealing with customized packers is even more challenging. Hence, we aim to perform classification without the need to unpack the sample. In addition, the system doesn't need to be evaluated on any packed goodware, because the problem of malware classification already assumes all of the samples to be malware. Finally, as the work focuses on malware classification, we didn't make any analysis of evasion mechanisms employed to evade detection.
For each malware sample, we compute not only a set of content-based features by relying on state-of-the-art mechanisms, but also we propose the extraction of powerful complementary statistical features that reflects the structure of portable executable (PE) files. The decision of not using more complex models like n-grams, sequences, bags or graphs, allowed us to devise a simple, yet effective, and efficient malware classification system. Moreover, we implemented an algorithm, inspired by the forward stepwise feature selection algorithm [153], to combine the most relevant feature categories to feed the classifier, and show the trade-off between the number of features and accuracy. To better exploit both the richness of the available information, in the number of the malware samples for training the classifier, and the number of features used to represent the samples, we resorted to ensemble techniques such as bagging [164].

We evaluated our system on the data provided by Microsoft for their malware Challenge hosted at Kaggle[3], and achieved 99.77% accuracy. The source code of our method

[3] https://www.kaggle.com/c/malware-classification

is available online[4].

1.4 Contribution

The success of machine learning for desktop malware classification motivated us to extend the mindset for Android malware. The first problem we look at is *malware detection*: operators of app markets wish to automatically check submitted apps for malicious or potentially harmful code to protect users. The second problem we are interested in is *family identification*: an important step of forensic analysis of malicious apps is to differentiate families of related or derived malware. Although there are many approaches for Android malware detection/family identification, the main difference of our proposed techniques in this thesis from the other proposed approaches for Android malware classification, is that we tried to model those functionalities of Android applications that can be misuesed in malware and no one has addressed it in the past. To this end, the contributions of this thesis are listed as follows:

(i) First, I provide a comprehensive overview of the vulnerabilities affecting the Android platforms and its users, the solutions devised so far to better highlight the security issues in Android platform. I cover all the relevant Android security issues emerged so far with a focus on Android application analysis, by organizing the presentation according to an original perspective that is reflected by the organization of the chapter. This perspective is further enriched by a number of original figures and tables that summarize the most relevant facts, that provide a clearer view of the Android Security landscape. (See Section 2)

(ii) Due to the importance of mobile malware, especially mobile botnets, we show how it is possible to effectively group mobile botnets families by analyzing the HTTP traffic they generate. To do so, we create malware clusters by looking at specific statistical information that are related to the HTTP traffic. This approach

[4] `https://github.com/ManSoSec/Microsoft-Malware-Challenge`

also allows us to extract signatures with which it is possible to precisely detect new malware that belong to the clustered families. Contrarily to x86 malware, we show that using fine-grained HTTP structural features do not increase detection performances. Finally, we point out how the HTTP information flow among mobile bots contains more information when compared to the one generated by desktop ones, allowing for a more precise detection of mobile threats (See Section 3).

(iii) As far as the network traffic generated by malware can be encrypted by libraries using SSL, our proposed approach in Section 3 based on HTTP traffic analysis might not be effective. An example of these libraries is Google Cloud Messaging (GCM). GCM is a widely-used and reliable mechanism that helps developers to build more efficient Android applications; in particular, it enables sending push notifications to an application only when new information is available for it on its servers. For this reason, GCM is now used by more than 60% among the most popular Android applications.[5] On the other hand, such a mechanism is also exploited by attackers to facilitate their malicious activities; e.g., to abuse functionality of advertisement libraries in adware, or to command and control bot clients. However, to our knowledge, the extent to which GCM is used in malicious Android applications (badware, for short) has never been evaluated before. Therefore, we do not only aim to investigate the aforementioned issue, but also to show how traces of GCM flows in Android applications can be exploited to improve Android badware detection. To this end, we first extend Flowdroid to extract GCM flows from Android applications. Then, we embed those flows in a vector space, and train different machine-learning algorithms to detect badware that use GCM to perform malicious activities. We demonstrate that combining different classifiers trained on the flows originated from GCM services allows us to improve the detection rate up to 2.4%, while decreasing the false positive rate by 1.9%, and, more interestingly, to correctly detect 14 never-before-seen badware

[5] http://www.zdnet.com/article/io-2013-more-than-half-of-apps-in-google-play-now-use-cloud-messaging/

applications. (See Section 4)

(iv) Although the proposed technique in Section 4 achieved promising results, obfuscation techniques like dynamic code loading or reflection can create a problem for those systems based on flow-analysis as well as those that have successfully relied on the extraction of lightweight syntactic features. Flow-analysis techniques are usually time-consuming and using naive features can be evaded by obfuscation techniques. To address the aforementioned challenge, we propose DroidSieve, an Android malware classifier based on static analysis that is fast, accurate, and resilient to obfuscation. For a given app, DroidSieve first decides whether the app is malicious and, if so, classifies it as belonging to a family of related malware. DroidSieve exploits obfuscation-invariant features and artifacts introduced by obfuscation mechanisms used in malware. At the same time, these purely static features are designed for processing at scale and can be extracted quickly. For malware detection, we achieve up to 99.82% accuracy with zero false positives; for family identification of obfuscated malware, we achieve 99.26% accuracy at a fraction of the computational cost of state-of-the-art techniques. (See Section 5)

Chapter 2

Overview on Android Security

2.1 Introduction

All of the security issues mentioned in section 1.2 have stimulated the research community, so that Android security rapidly became a hot topic in the past years. Studies on Android security have started since 2008 [109, 230], and since then, the number of papers in this field had consistently grown. Recently, some papers reported some overviews on the current state of the art on the security of mobile devices [50, 58, 106, 108, 118, 119, 122, 210, 257, 293]. Some of them focused on the threats models targeting mobile devices, while others focused on vulnerabilities of a specific platform. This chapter aims at providing a comprehensive overview of the vulnerabilities affecting the Android platforms and its users, the solutions devised so far, and future research directions. As a comparison with two recent surveys on the topic [119, 244], we cover all the relevant Android security issues emerged so far, by organizing the presentation according to an original perspective that is reflected by the organization of the chapter. This perspective is further enriched by a number of original figures and tables that summarize the most relevant facts, that provide a clearer view of the Android Security landscape. The rest of this chapter is organized as follows: We first briefly present the architecture of Android

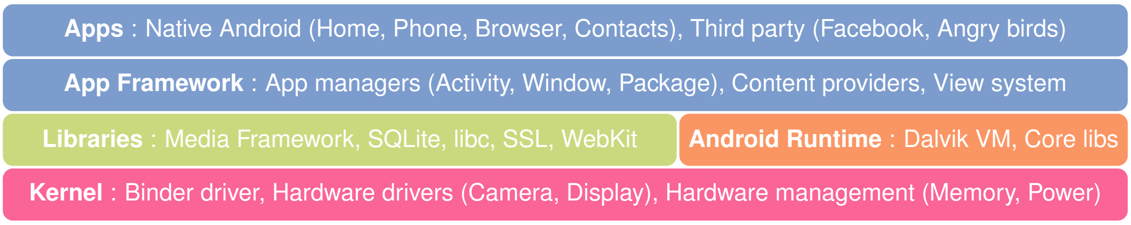

Apps : Native Android (Home, Phone, Browser, Contacts), Third party (Facebook, Angry birds)
App Framework : App managers (Activity, Window, Package), Content providers, View system
Libraries : Media Framework, SQLite, libc, SSL, WebKit
Android Runtime : Dalvik VM, Core libs
Kernel : Binder driver, Hardware drivers (Camera, Display), Hardware management (Memory, Power)

Figure 2.1: Android Architecture

OS layers, the Android App components, and the Android security features. Then, the security challenges related to the development of the Android OS are presented. We review the different mechanisms that have been proposed to analyze Android Apps afterwards. Furthermore, discusses the security threats related to the so-called Android fragmentation, i.e., the coexistence of Android devices with different Android versions.

2.2 Android background

Before we start our discussion on the security challenges of Android based systems, let's briefly review the Android OS, frameworks and Apps features. In the following, each component is described in detail.

2.2.1 Android OS

Android is a software stack for mobile devices and it is structured in several layers as shown in Fig 2.1.[1] The purpose of each layer is briefly described in the following paragraphs. We report the few details that are needed to support the description of the security issues of Android systems.

[1] https://developer.android.com/guide/platform/index.html

Kernel Layer

Android is built on top of the Linux kernel which contains core services that manage the device. It provides the permission architecture to restrict access to resources, supports file and network I/O, memory and process management. It also provides Android to communicate with low-level hardware components by supporting device drivers. In addition to the above generic services, it also includes Android-specific components such as power and memory management, because mobile devices have limited memory space, and must optimize battery use. Android also use a custom implementation of IPC called *binder*. The *binder* is a very efficient remote procedure call framework that allow fast interactions between activity and service components securely and concurrently. The *binder* is implemented as a device driver, and Apps use the `ioctl` system call to interact with the *binder*, so that monitoring `ioctl` operations allows tracing the interaction of an App with other elements outside of its sandbox.

Libraries Layer

Above the kernel layer, the library layer includes a variety of system (native) libraries which are typically written in C and C++. These libraries handle the core performance sensitive activities on the device as the surface manager library for updating the display, the media framework Library for managing various audio and video formats, Webkit for rendering and displaying web pages, SSL, SQLite, and other libraries. In addition, Android has its own system C libraries, `libc`, which implements the standard Linux system calls handling process and thread creation, mathematical computation, and memory allocation.

Android Runtime

At the same level of the Libraries, the Android run-time is the Virtual Machine (VM) for running Android Apps. The two main components of the Android runtime are the core Java libraries, and the Dalvik virtual machine (DVM), recently substituted by the

Android RunTime (ART). In this chapter, we use the term DVM because most of the works addressed it as DVM. Android Apps are written in the Java programming language. Android provides a number of reusable Java libraries to make programming easier for developers such as file I/O, concurrency, and web operations. After developers write their Apps in Java, the source code is compiled by the Java compile to produce the bytecode. Then the DX tool converts the Java bytecode to a single DEX bytecode file which is named `classes.dex`. This file is packaged with other App resources and installed on the device. The DVM is a custom VM (sandbox environment) that executes the code in `classes.dex`. The reason of designing a custom VM is that it was specifically designed to run in the resource constraint environment of mobile devices. Zygote is the first instance of the DVM that is created at boot time, and, when an App is launched, a new DVM instance is spawned by the Zygote process, so that each App has its own DVM instance. Zygote owns all the core libraries, and hence all Apps share the libraries owned by Zygote.

App Framework Layer

The App framework contains the reusable libraries that are emplyed by developers as building blocks for their applications, such as common graphical elements, audio multimedia management, etc. The *package manager* is a database that keeps track of all of the Apps installed on the device. It checks the permissions and stores information about the Apps. It also allows an App to contact other Apps, enable data sharing among Apps, and let one App to request services from other Apps. It also contains the *resource manager* which manages the resources of an App that are not modified by the compilation process, such as strings, graphical elements, and layout files. The *activity manager* coordinates the navigation between different activities of an App, while *content providers* are databases that allow Apps to store and share information to other Apps. The Framework Layer then comprises other App managers for all other functionalities supported by mobile devices such as the window manager, the location manager and the notification manager.

App Layer

Android comes with a number of pre-installed Apps subdivided into two sets, i.e., a set of Apps in common with all Android devices, and another set of Apps specifically designed by the device manufacturer. Android standard Apps include the home screen, the web browser, the phone dialer, Messages manager, Contacts Managers, etc. However, the user can install and use third party Apps that perform the same tasks, as well as other Apps performing various tasks, related to business, personal and entertainment purposes. Apps can be purchased and downloaded by a number of App stores, the official one being maintained by Google.

2.2.2 Android Apps

Android Apps are usually written in the Java language, and `classes.dex` contains the bytecode of an App. The code contains one or more of Android's App four fundamental components, such as Activity, Service, ContentProvider and BroadcastReceiver. Components communicate with each other in an App, and between different Apps through Inter Component Communication (ICC) mechanisms. Android Apps are basically protected against standard buffer overflow attacks due to the implicit bound checking implemented by the Java language [83].

—`Native code:` Developers can also build Android components using C/C++ languages, and the related code is called *native* code. The native code doesn't run in the DVM, but it is directly executed by the Linux kernel, thus directly accessing the kernel without using the framework layer. Consequently, resources on Android need to be protected both at the framework layer, and at the kernel layer. Since less than 10% of Apps use native code, the analysis of native code provide evidence for detecting malware and exploits [140].

—`Activity:` Activities provide the App GUI to the user, and capture user's interaction through the interface. An App UI consists of one or many activities, depending on the functionalities implemented by the developer, and only one of them runs in the foreground. Each activity consists of different UI elements, such as Button, EditText, CheckBox, etc.

—`Service:` Services have two main purposes. The first one is to perform long-duration background operations for specific tasks such as playing music, synchronizing data with cloud services, downloading large files from the Internet, etc. The second one is to support interaction, and share data with other processes.

—`BroadcastReceiver:` BroadcastReceivers listen and respond to broadcast events from the system. Events are represented by intents and then are broadcast.

—`ContentProvider:` ContentProviders define storage containers to share data across the Apps. They also handle inter-process communications so Apps can exchange data safely.

—`Intent:` In addition to the above main components, Intents are the other key component for App cooperation. Intents are an abstract representation of operations to be performed, and they are used to start activities, services or BroadcastReceivers.

—`Android package:` Android Apps, in addition to the bytecode, contain additional elements such as the `manifest` file, bitmaps, layout definitions, user interface strings, and libraries. The `manifest` file is a XML representation of contextual information about the App that is used by Android to run the bytecode. These information include the package name, components, permissions, and linked libraries.

2.2.3 Android security features

Google released Android 1.0 in September 2008. This version included some basic security mechanisms such as discretionary access control (DAC), App sandbox, mandatory access control (MAC) permission model, secure ICC, and App signing [111]. DAC is used to control the access to files by process ownership. The sandboxing mechanism isolates Apps from system resources, and from the execution environment of other App running in the system. The permission mechanism protects sensitive interfaces, and also enforces mandatory access control on ICC calls. Finally, all Android Apps are digitally signed with a cryptographic signature to verify the origin of Apps, and to establish trust relationships among them. The Android versions that followed included additional security features, while known vulnerabilities have been corrected. In addition, security enhancement have been also included in the official Google Play market so that App are added to the the market after passing some security checks like Google Bouncer. The most recent report by Google [219] shows that less than 1% of all devices had a Potentially Harmful App (PHA) installed and fewer than 0.15% of devices that download only from Google Play had a PHA installed. The most relevant Android security features are shown as a time-line in Fig 2.2.

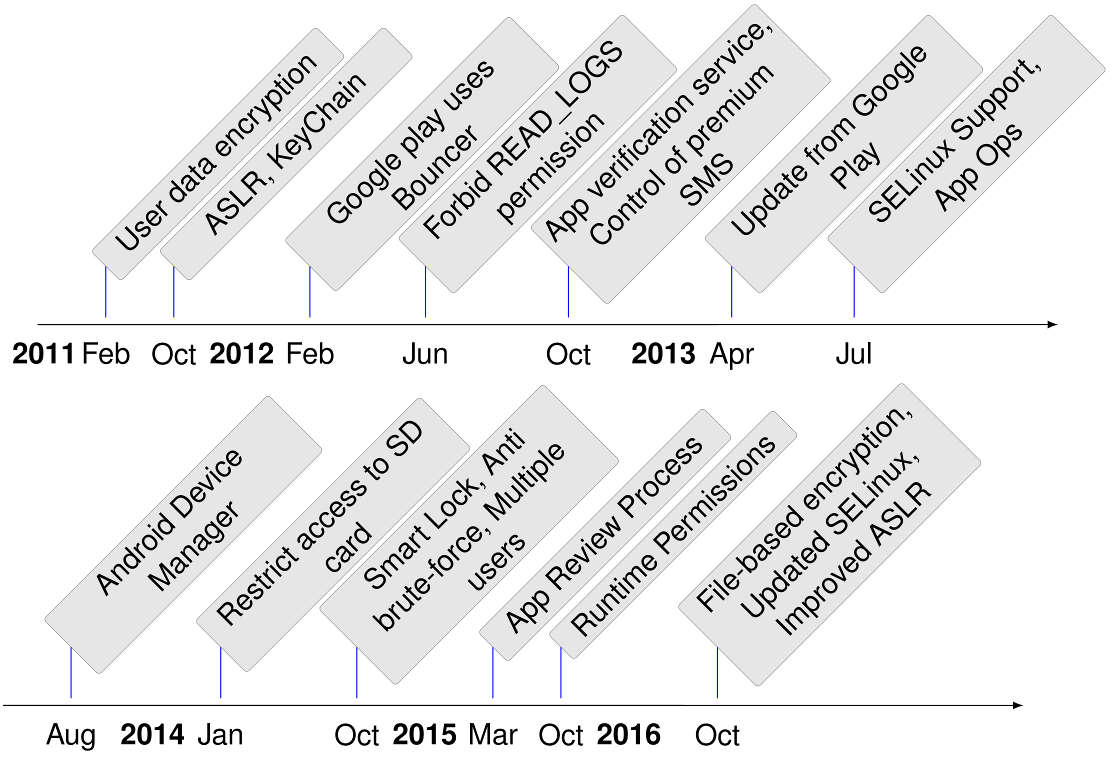


Figure 2.2: Timeline of Android Security features since 2011

February 2011: Data encryption has been an option since Android 3.0 [20]. Some users are unaware of this feature, while other users avoids it because it slows down the processing time of the device. The interesting issue is that cryptographic keys are not stored on the device, so they can't be shared with law enforcement agencies.

October 2011: Android includes many techniques to protect against exploitation of memory corruption. One of the most important methods is Address Space Layout Randomization (ASLR), that was included in Android 4.0 to protect from attacks based on the knowledge of process layout in memory. [20]. Android 4.0 also includes a KeyChain service [7] that provides a system-wide credential storage to access private keys and their corresponding certificate chains.

February 2012: Google added a new security layer to the Google Play store called Bouncer [133]. Google Bouncer provides automated scanning of Apps submitted to the Play store for potentially malicious software. There is no detailed information by Google on the internals, but some researchers dissected it by uploading an App to Google play, and then establishing a connection between the App and a C&C server, getting some information from it [192].

June 2012: READ_LOGS is a permission which is required for an App to be able to read the logs. Apps can't get this permission since Android 4.1, unless they are part of the firmware because Log entries can contain the user's private information [209, 252].

October 2012: Android 4.2 was released with an App verification service which utilizes signature-based detection techniques which is not very effective against unknown malware [156]. Another feature in this version is a notification while an App attempts to send premium SMS which might cause additional charges. User can choose whether to allow or block the App to send messages, a very helpful feature because many malware families steal money from the victims by sending premium SMS messages [293].

April 2013: Google changed the content policy for Google Play : "App downloaded from Google Play may not modify, replace or update the binary code of your APK using methods other than the update mechanism of Google Play" [137]. This policy improved the overall security of Android because it increased the difficulty for an App to secretly download a malicious software as it was an update.

July 2013: Android 4.3 was the first version of Android to fully include SELinux support, contributed by the SE for Android project. Security-Enhanced Linux is a mandatory access control system built into the Linux kernel to help enforce the permissions, and to attempt to prevent privilege escalation attacks (i.e. an App tries to gain root access on your device) [135, 176, 236] . A detailed permissions manager called App Ops was introduced in Android 4.3, that let you disable permissions [40]. Google removed this privacy control in Android 4.4.2 and said "The feature had only ever been released by accident and it was experimental. It could break some of the apps policed by it.".

August 2013: In December 2013, Google presented the Android Device Manager (ADM) as an App to allow users to remotely locate their device or erase their data from the device through a web interface. The App is available on the Google Play store and it runs on Android version 2.3 and higher [134] .

January 2014: In Android 4.4.2, Apps can no longer write on any part of external storage media like SD card [136]. Although SDFix App with root access can restore this ability, apps are generally only allowed to write to a special, "App specific", folder on the External SD card [190]. The only warning is that if you uninstall the App, everything that is stored in the "App specific" folder will be deleted.

October 2014: Android 5 was released in October 2014 with some security improvement [138] in many domains such as protection against memory-corruption vulnerabilities, encryption, access control with SELinux, cryptography for HTTPS. In addition, it provided a new feature, which is called Smart Lock, that allows devices to be unlocked automatically when they are close to another trusted device or being used by someone with a trusted face. Also starting with Android 5.0 [219], the user passwords are protected against brute-force attacks, as well as providing multiple users was provided on phones including a guest mode, which had been introduced on tablets in Android 4.2.

March 2015: Google decided to manually review submitted Apps to the Google Play [139]. This new process involves a team of experts who are responsible for identifying violations of Google’s developer policies earlier in the app lifecycle.

October 2015: The major security enhancement for Android in 2015 is enabling permissions during runtime. After a number of research papers addressed finer-grain access control mechanisms, Google finally announced in Google I/O 2015 that the next version of Android, namely Android 6 (Android M), will officially come out with granular permission control.

October 2016: Google enabled encryption at the file level, instead of encrypting the entire storage area as a single unit, which can help to better isolate users' profiles. In addition, Google announced some improvements and updates for ASLR and SELinux to make the system more reliable against attacks. All of these changes affect Android 7 (Android N).

2.3 Android OS security challenges

The Android kernel and the middleware layer offer to Apps the mechanisms to interact with the other Apps on the same device, and with external services. It turns out that if a vulnerability is found in these layers, it can cause critical issues and severe problems to users. To date, the main security issues caused by misusing two security features of these layers are privilege escalation and information leakage. In this section, we will detail the vulnerabilities that may lead to privilege escalation and information leakage, and we will also describe how Android can be protected from those potential threats.

2.3.1 Privilege escalation

The most dangerous Android vulnerability categories, according to the CVE score, are *code execution* and *gain privilege* [86]. Attackers usually start by executing buffer overflow attacks to place their code, and finally escalate their privilege to get root shell. A number of root exploits have been devised for the Android platform, and in Table 2.1 we listed the Top 10 in terms of their popularity [16, 140, 150, 270]. Apps that exploit these kernel layer vulnerabilities, pose the highest risk for the device as they gain superuser privileges. All of the existing malware containing root exploits uses third-party native code. Unfortunately, even the most recent versions of malware analyzers typically ignore third-party native code because of its difficulty compared to system libraries or Java code.

Another issue, is that of patching the vulnerabilities, as this task is up to the manufacturers that usually doesn't provide updates and patches for all the handsets, although Google regularly patches the vulnerabilities [79]: To provide a solution to this prob-

Table 2.1: Root Exploits

Year	Name	Target	In Malware
2009	Asroot (Wunderbar) [113]	Linux kernel [89]	✓
2010	Exploid [114]	init [90]	✓
2010	RATC [3]	ADB daemon	✓
2011	GingerBreak [9]	vold [93]	✓
2011	ZimperLich [10]	zygote	✓
2011	Levitator [158]	PowerVR driver [92]	×
2011	KillingInTheNameOf [5, 8]	Ashmem [91]	×
2011	zergRush [102]	libsysutils [94]	×
2012	Mempodroid [105]	Linux kernel [95]	×
2014	Towelroot [150]	Linux kernel [96]	✓
2015	PingPongRoot [266]	Linux kernel [97]	✓

lem, PatchDroid has been proposed as a prevention-oriented system to distribute and apply device-independent security patches for vulnerabilities in both native and managed code [183]. Another tool aimed at detecting root exploit is PREC (Practical Root Exploit Containment), a response-oriented system to dynamically identify system calls from third-party native libraries, execute them within isolated threads to detect, and stop root exploits with high accuracy and low false positive [148].

Permission escalation

Another kind of privilege escalation vulnerabilities is permission escalation that occurs both in Apps and in the middleware layer. Permission escalation attacks are divided into tow categories, namely confused deputy attacks, and collusion attacks. In confused deputy attacks [146], which are also called transitive permission usage, a malicious App try to communicate with other benign Apps to access critical resources without any explicit request to the corresponding permissions. In fact, a malicious App can use a benign application as a service to escalate their privileges, if the permissions of the benign application have not been carefully configured. In the alternative case, i.e., collusion attacks, two malicious Apps collaborate each other to integrate their permissions and perform a malicious behavior [180]. These attacks can be traced back to fundamental flaws in the Android sandbox model [66, 98, 118, 124, 180]. In fact, the Android permission system does not support the identification of transitive requests or collusion attacks.

[269] also showed that permission escalation may happen during the Android update process. They reported a vulnerability in the package manager service of Android that allows a malicious App to define a set of permissions in lower version of Android, and wait until Android is updated to the new version, and benefit the permissions on the new system. They could also contaminate data to steal sensitive information, cause denial of service and replace some new system Apps during the update process.

2.3.2 Information Leakage

Information leakage is a vulnerability that may reveal sensitive data to adversaries thereby leading to an attack. One of the Android services that was found to exhibit information leakage problems was the Android logging service. If an App has set the permission read_log permission, it can read some sensitive information such as Geographic location and web requests in the logs [174]. Google removed this permission as we showed in Section 2.2.3. Another important source of leakage is the screen of the device. Screenmilker is an approach that captures screenshot on Android devices without any privileges by exploiting a vulnerability in the ADB proxy [170]. The vulnerability grants privileged capabilities to 3rd party Apps and it can be exploited by malicious Apps to extract sensitive information from screenshots.

Side channel attacks

Side channel attacks refer to any attack based on the information flow within the Android device hardware/software stack, that lead to information leakage. Some of these information can be accessed from the Linux `proc` file system, or from the sensors. A classification of this kind of attacks is shown in Fig 2.3. One of these attacks, *Memento*, investigates the information leaks in Android that can be carried out by analyzing shared memory usage data [154]. [290] is another approach that uses the `proc` files to exploit public information like social networks background data to retrieve the identity or the location of the users without requesting any special permissions. In a recent paper, it was shown that it is possible to verify the state of the UI without needing the exact values of the pixels of the screen [76]. They were able to hijack sensitive UI states to steal private inputs such as passwords, pictures from the camera, and they also showed how an attacker can monitor and analyze user behavior. They concluded that UI state leakage is possible, and that an unprivileged App can track another App's UI states while it is running. They also show one example in which the the state of the Log-in UI has been caught via shared-memory, CPU and network activity side channels. Finally, they showed how the confidentiality and integrity of the GUI content can be broken. A prominent example of that is the smudge attacks that allows understanding the visual Android password pattern by observing the oily residues remaining on touchscreens after a pattern is entered [54].

- `Sensors Information Leakage`

Smartphone sensors, the speakers and cameras are other channels that convey sensitive information that can be exploited by attackers. Some people might think that some sensors like the Accelerometer, the Gyroscope and the orientation sensors are not privacy sensitive, but different works [197, 272] showed that they can be turn out to be dangerous channels. TouchLogger [71] is one of the major works showing how internal sensors can be exploited by a keylogger software based on capturing rotations from motion sensors. The keylogger could infer keystrokes with 71.5% accuracy. While the authors didn't evaluate the behavior on the whole keyboard, and they also didn't verify if their method is user and device Independent, in a following work [72], they also evaluated soft-keyboard input on tablets and smartphones. Another approach was presented in *Gyrophone* [181], where authors could extract the speech from Gyroscope sensors by using signal processing and machine learning techniques. *Soundcomber* [229] is another system that shows how a malicious App can exploit the speaker to learn the difference

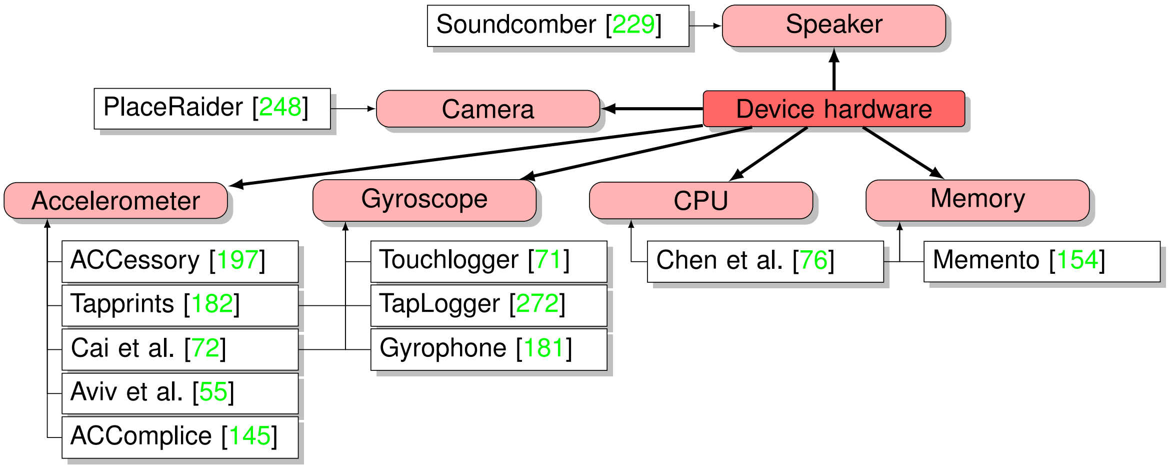


Figure 2.3: Side channel attacks

between general chatter, and tone dialing, to effectively extract the phone numbers a user calls. *PlaceRaider* [248] focuses on the camera, and constructs 3D models of indoor environments by using the phone's, camera, Accelerometer and Gyroscope. Cai et al. [73] proposed a framework with several promising methods to defend against sensory malware. Summing up, malicious applications can be designed to capture data form smartphone sensors that can reveal a large number of information related to the data captured by the sensors, and to the general behavior of the user. As far as just few papers [73, 273] addressed this issue, proposing an ideal protection mechanism should be considered.

2.3.3 Security policies and mechanisms

While Android has improved the basic security features over the years as described in section 2.2.3, that prevents attacks that undermine the correct operations of the operating system, nonetheless attacks can still be carried out using the same mechanisms used by benign applications to communicate with each other, and with the outside world, Hence, the research community is quite active in proposing additional security policies and mechanisms for Android in both the middleware and the kernel layers. In this section, we focus on security extensions in the kernel and the middleware layer.

Privacy leakage protection

Privacy leakage refers to a type of information leakage that involves the transmission of sensitive data out of an Android phone. However, if the transmission is by user intention, it does not necessarily translates into a privacy leakage, while in all other cases it is more likely that a privacy leakage took place. One of the techniques used to detect privacy leakages is by dynamic taint analysis, which tracks information dependencies from an origin which is called taint source. Taint sources comprise sensors like GPS, information databases such as the SMS archive, and the Device ID such as the IMEI. On the other side, taint sinks refer to the interfaces that can be used for leaking privacy data, such as the use of the network connection to send out sensitive data. There are several approaches for taint analysis, that are usually categorized into different tracking categories such as variable-level, method-level, message-level and file-level. Variable-level tracking refers to storing and propagating taint tags on variables by modifying the DVM stack which contains variables. Method-level tracking provides information about native method invocation, while message-level tracking refers to tracking communications between apps via IPC mechanisms. Finally, file-level tracking refers to tracking data communications between apps and storage media.

TaintDroid [107] is the first and prominent example of dynamic taint analysis that tracks data flows across the Apps and the Android OS. It modifies the DVM interpreter to track variables, thus requiring an intensive instrumentation of the OS to get a good performance. One of the main benefits of Taintdroid is tracking taints through reflective method calls. The main shortcoming being code coverage in the case malicious Apps understand that they are executing in an analyzing environment. Taintdroid can only monitor data flows (explicit flows), while it can not monitor control flows (implicit flows). In addition, it is implemented in the middleware layer, thus making impossible to track native code. AppFence [149] and CleanOS [247] leverage Taintdroid to track sensitive data for different purposes. AppFence aims at providing protection against evicting sensitive data from the phone, while CleanOS aims at protecting users against disclosure of private data in the case the phone is stolen. CleanOS encrypts phone's data when the phone is idle for a specified amount of time, and then sends the encryption key to a trusted cloud service immediately. The main problem of CleanOS is that it can't prevent leakage through OS or I/O channels. D2Taint [143] and DataChest [295] are other two extensions of TaintDroid. D2Taint tracks the information coming from Internet sources, while DataChest tracks and protects unstructured data coming from users. We summarized the proposed approaches in Table 2.2 where they have been sorted by the date of publication. All of these approaches are implemented in the middleware layer. The column labeled *Tracking* shows if the system performs dynamic taint analysis, and it it labels the sensitive data, while the column labeled *prevention* shows if the system aims to prevent privacy leakages, and the last column shows the main drawbacks of each system.

Some other approaches have been proposed to limit untrusted Apps to access specific sensitive data. Apex [187] enforces some rate limiting like the number of sent messages during run-time, as well as enabling or disabling specific permissions during install-time. Similar to Apex, MockDroid [59] and TISSA [297] allow users to better control the fine-grained accesses to specific resources or permissions during runtime.

Access control mechanisms

There are different access control mechanisms to mitigate privilege escalation attacks as shown in Table 2.3. Saint [196] is a mandatory access control solution that helps developers to define policies and protect Apps from being exploited. Policies are checked and stored at install-time, and they are enforced during run-time. App developers are allowed to choose the more suitable App policy solution, and determine how their apps can interact with the other Apps installed in the system and vice versa. As malicious Apps that perform privilege escalation through root exploits can bypass the middleware layer, and use the system resources (e.g. radio or services like SMS, Call) without permissions, access control mechanisms are required at both the user-space and the

Table 2.2: Privacy leak protection mechanisms

Name	Tracking	Taint source	Prevention
Response-oriented mechanisms			
TaintDroid [107]	✓	32 APIs (IMEI, Phone Number, Location)	×
Appfence* [149]	✓	32 APIs (IMEI, Phone Number, Location)	✓
CleanOS* [247]	✓	User-provided	✓
D2Taint* [143]	✓	Internet	×
Vetdroid* [283]	✓	IMEI, Phone Number, Location, Network State	×
DataChest* [295]	✓	User-provided	✓
Prevention-oriented mechanisms			
Apex [187]	×	—	✓
MockDroid [59]	×	—	✓
TISSA [297]	×	—	✓
Porscha [195]	×	—	✓

* These systems leverage TaintDroid

– All of the systems are implemented in the middleware layer

Table 2.3: Access control systems against misuses

Name	Implementation		Main Goal
	Middleware	Kernel	
IPC Inspection [124]	✓		Prevention of confused deputy attacks
Quire [104]	✓	✓	Prevention of confused deputy attacks
XManDroid [65]	✓		Prevention of confused deputy and collision attacks
Sorbet [126]	✓		Prevention of confused deputy attacks
Saint [196]	✓		Finer-grained access control
CRePE [82]	✓		Defining context-related policies
Aquifer [184]	✓	✓	Defining UI workflow policies
TrustDroid [67]	✓	✓	Mandatory access control
SEAndroid [236]		✓	Mandatory access control
Flaskdroid [68]	✓	✓	Mandatory access control
RGBDroid [200]		✓	Kernel-level privilege escalation mitigation
PREC [148]	✓		Kernel-level privilege escalation mitigation
L4Android [165]		✓	Virtualization against kernel-based attacks
AirBag [261]	✓	✓	Application isolation against AOSP attacks

Kernel levels. XManDroid [65] and TrustDroid [67] add some additional enforcement in the Kernel Layer by modifying the Mandatory Access Control framework in Linux. SEAndroid [236] is another MAC framework by leveraging SELinux for Android kernel. Afterward, Google introduced SELinux in Android 4.3 to mitigate the damages of root escalation attacks. Bugiel et al. [68] proposed a policy-driven way for extending the Android security architecture that works on both the Kernel and the Middleware layer. They developed a policy language as an extension of SElinux enforcement language, and presented a number of related use-cases. They developed an App called *object manager* to instrument system components in the middleware layer and kernel resources in the kernel layer. The *object manager* can assign security levels to objects, and enforce fine-grain access control mechanisms on each object. In each layer, all of the policies are stored on a security server where it manages the policy rules and contains the access decision logic. They leverage SEAndroid kernel level modules for the kernel layer, and extend this enforcement into components of the middleware layer such as the Services, the Package Manager, and Content Provider such as Contacts Provider. They also developed a user policy App to update policy rules. For example, it is possible to define fine-grained access policies to limit the access of social network apps to selected fields of contacts such as the email or the phone number. Differently from Prec, which was discussed in section 2.3.1, RGBDroid [200] is another response-oriented system against kernel-level privilege escalation attacks. It analyzes file access patterns on Android platforms by hooking system calls. It uses a white list which sustains the list of programs that can run with root privileges. It denies any resource access request by programs which are not in the white list. It also uses a critical list that contains the list of system layer resources that even a process with root privilege cannot modify. Quire [104] and CRePE [82] are two approaches to mitigate permission escalation. L4Android [165] and Cells [44] propose to increase the security by virtualization techniques, allowing multiple virtual smartphones to run side-by-side on one single physical device. In a more recent work, AirBag, supports a decoupled and isolated runtime environment based on OS-level virtualization [261]. AirBag aims to protect the OS from malicious untrusted

Apps by decoupling an App Isolation Runtime (AIR) from the native Android runtime, which contains Java and Native Libraries, DVM and App framework.

Embedded UI security

Embedding third-party UIs are common in Android apps for advertisement, and they can be used for other purposes, such as embedding maps, camera preview, social network follow buttons, etc. In Android, the UI composes of View and ViewGroup objects, organized in a single view-tree structure. By default, Android does not provide secure UI components isolation, and libraries run in the App's context. However, in the UILayoutTree structure, you should care about both malicious child and parent elements. There are many attacks target embedded UI such as input eavesdropping, and clickjacking. Some works just focused on advertising libraries such as AdDroid [202] and AdSplit [235] by isolating advertising libraries from host Apps, and some others vet embedded UIs in general. For example, LayerCake [222] provides a modified version of Android that securely support Inter-App embedded user interface. It allows an activity in one App to securely embed an activity from another App. LayerCake relies on modifications of the Activity Manager and the Window Manager to reach the goal. This approach consists of three steps. 1) Separation of each embedded UI from its own process by modifying the Activity Manager to expose a new UI element. It causes multiple activities to be in the foreground, and also takes into account parent-child communication. This prevents direct UI manipulation. 2) Windows (UI layout trees) must be separated, and nested UI trees must not be created. This goal has been achieved by modifying the Window manager. This prevents input eavesdropping and DoS attacks. 3) Some additional security measures like preventing clickjacking, handling size conflicts (e.g. malicious App create a very small or out of window camera preview), and preventing ancestor redirection (a malicious App tries to open a new top-level activity) are also taken into account by this module.

Other security mechanisms

- **Misuse detection:**
  Kirin [110] is an extension of the Android middleware to protect users against misuses by specifying additional access policies. Kirin denies App installation if the App requests potential dangerous permissions patterns. It looks for hard-coded dangerous combinations of permissions to warn the user about potential malware

- **Memory protection systems:**
  ASLR is a protection system against memory attacks, and it has been recently implemented in Android as we discussed in section 2.2.3. However, Lee at al. [166] showed that the Zygote process creation model weakens the effectiveness of ASLR because all App processes are created with largely identical memory layouts. They designed different attacks that bypass the weakened ASLR, and execute attacks based on Return-Oriented Programming (ROP) for Android. They also designed and implemented a secure replacement of Zygote called Morula.

2.3.4 Android fragmentation

One of the most prominent security issue of Android is the so-called *fragmentation*, as different versions of Android, with different maturity levels in terms of the security features, are currently in use. This phenomenon can either be seen as a feature that strengthens or weakens the Android ecosystem. Android fragmentation refers to the variety of Android platform versions currently used. At the time of writing, 22 official Android versions are available, each version being usually customized by device manufacturers (18,796 distinct devices were reported in [70]). The customization process usually do not involve the security features of Android, nonetheless vendor customization will inherently impact the overall Android security assessment when, for example, their own drivers are written, and standard Apps are customized. Another issue related to fragmentation and to vendor dependencies, is the application of patches regularly released by Google. For many devices, the manufacturer does not provide any update of the release, or they are applied at a slower pace. On the other hand, iOS and BlackBerry OS does not suffer of this issue, as the developer of the OS is the same as the manufacturer of the devices, so that any update of the OS is notified to all the devices, and in a short time window the vast majority of devices run the latest version of the OS.

SEFA [263] is one of the few approaches that investigate the security issues related to Apps customization. They look at thousands lines of code from ten different Android smartphones. They could verify their security flaws and determine each App's resource and permission usage, and their vulnerabilities. They perform provenance analysis to classify each App of a factory `Image` (source files) into three categories such as apps originating from the Android Open Source Project (AOSP), apps customized or written by the vendor, and third-party apps that are simply bundled into the stock `Image`. Then, they analyze permission usages of pre-loaded apps to identify overprivileged ones that unnecessarily request more Android permissions than they actually use. And, finally, they detect buggy apps that can be exploited to mount permission re-delegation attacks or leak private information.

In the aspect of hardware customization, ADDICTED [291] is a system that compares the protection level of drivers on customized phones to the corresponding version in Android. They showed that there could be a security hazard if there is a difference between the Android version and the customized version, so that an App which is not granted any root privilege, nor the permission to access the considered device, can use the flaw to access the device. For example, if camera device's files are publicly readable, any App can read it but in fact, it should be accessible only by camera group. At first, they developed a system for the automatic identification of protected devices' files, as it is not a trivial task because the name of the files can be different from the standard name of the device (e.g., a NFC device file is usually named /dev/pn544). So, they created a test runner App, and monitor its behavior with respect to the use of the device files, the dynamic analysis of its execution, and the use of system calls. After that, they could find out the device nodes associated to each device. Finally, they compare the vendor's customized files permissions with the corresponding AOSP, and, if there is any difference between permissions, it means that there is a security flaw.

2.4 Android Apps security challenges

Although Android has security features built into the operating system that significantly reduce the frequency and impact of App security issues, there are a number of important security issues that have to be taken into account during Apps development. On one hand, in order to attain a thorough analysis of the App behavior, the effects of each App instructions should be observed at the middleware and the Kernel layers. It should be clear from the above discussion that the implementation of such an observation mechanism would require modifications of the source code of the Android OS, that of course would disrupt the integrity of the execution environment. Therefore, rather than instrumentation, App behavior is usually captured either by static analysis, or by virtual machine monitoring. In the following sections, we will address the issues related to secure coding, and App analysis.

2.4.1 Secure coding

Android programming requires a deep knowledge of the security features of Android, and the security implications of each App component [51]. First of all, the usage of permissions should obey to the principle of least privilege. Furthermore, developers should be aware of the available security capabilities in Android such as communications via HTTPs, and cryptography APIs which can help their applications to be safer. On the other hand, the Android components such as services, SQLite, and webviews have to be used in a safe way. Moreover, developers should also consider using some abstractions during coding to avoid propagation of mistakes in Apps from other developers. For example, defining a logical security layer can lead to catch security flaws during compile time rather than run-time. Thus, developers should take care of different kinds of attacks such as data injection, information or capability leakage from other Apps. In the following, we overview some of the most relevant categories.

Data Injection: Apps or humans may provide malicious information for other apps. One of the protection mechanisms by developers to mitigate the risk of processing false information is checking the validity of input data. For example, it has been shown that an App can receive a malicious link from other apps every time a piece of text is pasted [117,282]. It has also been shown that Android Apps may suffer from well-known attacks such as SQL injection.

Information leakage: An App may have access to other App's data whereby can steal data either from the storage area or from the memory level, either directly or indirectly by an intermediate App. One of the issues that has been addressed since Android KitKat is the use of SD card as saving location. On lower version of Android, SD card can be used to store any type of data, thus making it possible to accidentally divulge sensitive data. Since Android Kitkat, apps can no longer write on every parts of SD card, and its use is limited to storing multimedia files and apps that does not manage sensitive data. For example, Skype for Android stored sensitive user data without encryption in Sqlite3 databases with weak permissions, thus allowing other local Apps to read user IDs, contacts, phone numbers, date of birth, instant message logs, and other private information [85]. Another example is divulging information through `logcat`, that has been observed for the Facebook App [209].

Capability leakage: Each App can share data or capability outside of its sandbox through IPCs, thus making it possible for malicious apps to escalate their capabilities through this functionality. These issues in Android programming emerge as vulnerabilities of the platform, as we discussed in section 2.3.1, if developers don't carefully define some policies to protect themselves against them. For instance, if an App has Internet access and accepts intents with an `url` for benign purposes, some malicious apps ,which have no internet permission, may exploit it to download their payload part. The simple solution for these cases is either to validate the information and the sender of intents, or to check if the caller has the specified permission or not.

Physical Access: Saving data privately is not a complete solution, as people that has physical access to the device may exploit stored data with different kinds of attacks such as rooting the device, or plugging in into a computer.

User mistakes: Other sources of attacks are similar to those against desktop computers, such as phishing. Phishing can be more effective on mobile devices as the limited screen size compared to desktop PCs makes it easier for a phishing website to mimic the legitimate one. In addition, the space for the address bar in webview is very limited, and very often it is hidden to the user thus making the detection difficult [123].

2.4.2 Android Malware

Mobile malware refers to any unwanted App that is used to perform an unauthorized and harmful action on mobile devices. One of the main goals of mobile malware has been to steal money from the user by issuing Calls and SMS to premium numbers. In addition, mobile malware abuse permissions, manipulate data, and access sensitive data, and then transmit them to the remote servers. They also exploit different vulnerabilities such as App-level privilege escalation, as in confused deputy attacks, and kernel-level vulnerabilities to gain root privileges. Multiple Android infection channels are exploited by malware writers such as QR codes [161], downloads from the market, downloads from the Internet, cross-platform malware that propagates from desktop clients to mobile devices [173, 245], and SMS malware. Although several works addressed these issues, as the number of Android malware is not decreasing, the effective analysis of Android app for malware detection still remains an open problem [108, 223, 293].

2.4.3 Android App analysis

The main purpose of App analysis is assessing if it is benign or malicious. The analysis of benign Apps, i.e., Apps that do not perform malicious activities, may reveal vulnerabilities that might be exploited for malicious activities, vulnerabilities caused by developers that neglect to securely design and implement their Apps. These Apps are usually referred to as Potential Harmful Applications (PHA). In this case, code review

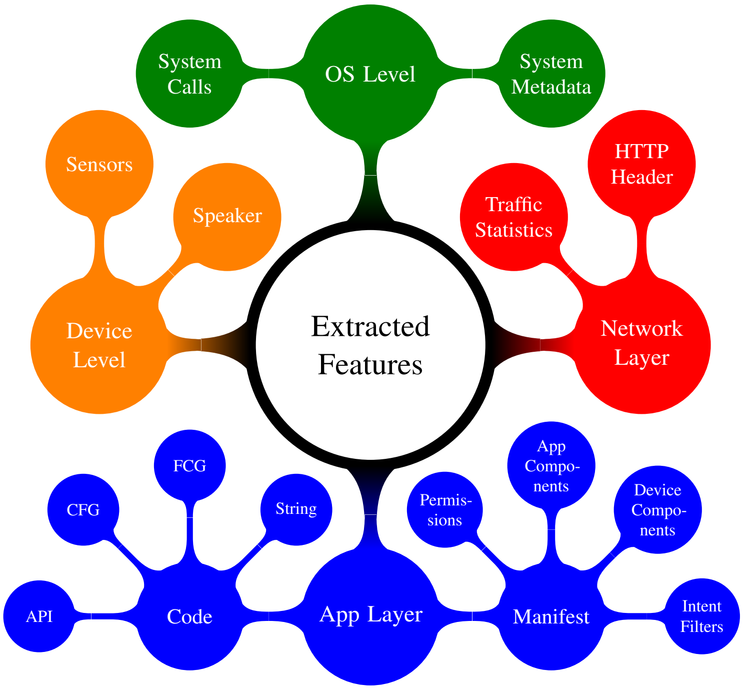


Figure 2.4: Classification of techniques for the analysis of Android Apps

and analysis is performed to discover unwanted vulnerabilities. On the other hand, when the malicious nature of an App is detected, then it is assigned to one of the classes which malware is usually categorized in. In order to understand each App, different kinds of information can be extracted from App or System (see Figure 2.4). Some of the examples are monitoring HTTP traffic in the network layer, considering system calls in the OS level, monitoring sensors in the device level, and extracting various structures such as Control Flow Graph (CFG) from the byte code in App layer.

Apps are analyzed according to a number of techniques that can be broadly subdivided into two groups, namely, static analysis techniques, and dynamic analysis techniques. Static analysis techniques comprise all techniques that do not execute Apps, neither on real devices, nor in emulation environments. Thus, from the point of view of the tools and equipment needed, static analysis can be carried out easier than dynamic analysis, as the latter requires an appropriate execution environment in order to extract the behavior.

Static analysis collects behavior information from all the components of an Android

App, by analyzing the elements in the manifest file, and decompiling the Dalvik bytecode. Soot [17], Dare [12, 193], and dex2jar [15] are some examples of decompilers for converting the Dex format into java format. The major problem for static analysis techniques is to cope with obfuscation (see Section 2.4.6) and packing techniques, that are increasingly used by malware writers in order to prevent or mislead the extraction of behavioral information through static analysis. The use of reflection mechanisms to invoke functions, or the use of native-code libraries, are two examples of coding approaches that may affect the correct and complete understanding of the behavior at runtime.

On the other hand, dynamic analysis techniques collect behavioral information by running the Apps. Two main approaches for dynamic analysis are followed, i.e., by instrumentation, which requires modifications of the Android OS (see Section 2.3.3), or by virtual machine monitoring techniques. One of the first dynamic analysis systems for Android App that relied on virtual machine monitoring was the ARE system, which was offered by the Honeynet project [6] for profiling Android Apps. The main advantage of dynamic analysis systems is that they are resilient against obfuscation of the bytecode, while they can be mislead by runtime behavior diversification if the App is designed to exhibit different behaviors depending on the execution environment.

So, while in principle dynamic analysis can be less complex than static analysis as, for example, the real sequence of system calls are collected, on the other hand, the tools required to collect such a behavior might be more complex to be developed. In the case of dynamic analysis by virtual machine monitoring, a two-step analysis is required. The first step requires a deep understanding of the Android specific behaviors related to its phone functions such as SMS, and CALL, while the second step requires the understanding of the typical OS interactions, like opening a file or a socket. Opposite to instrumentation techniques, whose presence can be detected by malware, VM-based analysis systems are more transparent and can be hardly detected by malware. An overview of the online services and offline tools that are available for static and dynamic analysis is reported in Table 2.4.

Table 2.4: Open source tools and services for Android App analysis

Name	Analysis type			Availability	
	Static	VM-based	Instrument	Web service	Open source
Androguard [11]	✓				✓
Copperdroid [14]	✓	✓		✓	
Apktool [4]	✓				✓
FlowDroid [25]	✓				✓
Andrubis [21]	✓		✓	✓	
A5 [18, 19]	✓	✓		✓	✓
droidbox [24]		✓			✓
Mobile sandbox [27]	✓	✓		✓	
Taintdroid [32]			✓		✓
smali [30]	✓				✓
Dexter [23]	✓			✓	
Droidscope [13]	✓	✓			✓
virustotal [35]	✓	✓		✓	

As we mentioned before, there are two main purposes for App analysis, such as vulnerability analysis and malware detection, that are described in the following section, and the most noticeable App analysis techniques are compared in Table 2.5.

2.4.4 Purposes of App analysis

Vulnerability analysis

App vulnerabilities are identified as the *number one* security threat. Some developers don't have enough security knowledge, or they simply neglect to secure their App. For example, FireEye [284] found that 68% of 1000 most-downloaded free Android Apps are susceptible to MITM attacks based on SSL vulnerabilities. Hence, a lot of researchers analyze Apps to find potential securities vulnerabilities. Google itself is committed to this task via its App analyzer framework that analyze all Apps distributed through Google Play, and also other Apps distributed by other markets as soon as they are installed on Android devices [219]. MalloDroid [116] is one of the approaches that relies on static analysis to find misuses in communications via SSL by Android Apps to find specific programming errors. Android Apps vulnerabilities also include SQL injection, which is one of the major vulnerabilities affecting web application. ContentScope [294] showed that SQL injection attacks can be used to extract some private data from unprotected Content Providers. Another threat for Android devices is related to advertisement libraries, that are used mainly by free Apps. AdRisk [141] is a system specifically designed to examine advertisement libraries by static analysis, to see if any sensitive information is uploaded to remote advertisement servers, or if any untrusted code is executed.

In addition of more general vulnerabilities like SSL and SQL injection, that are not specific to Android Apps, there is a vast literature addressing Android specific threats. In particular, it has been shown that vulnerabilities are not just related to individual components, but many of them can be discovered and exploited through the analysis of the interaction between Apps through the Inter-Component Communication (ICC) mechanisms. Android ICCs are based on *Intents* and *Intent Filters*, and the components that communicate are protected by some permissions. There are three main risks related to the inter component communication models, as insufficiently protected components may leak capabilities (e.g., a malicious component can exploit other components to send SMS to premium-rate numbers), a malicious component can intercept intents, and steal information that is hold by intents, and, finally, Apps can communicate to each other to perform a malicious task, which is called App collision (e.g., if the malicious activity requires GPS and Internet permissions, it can be performed by two Apps, let's say App A and App B, that work concurrently, App A having GPS permission, and App B having Internet permission). As we explained in section 2.3.3, some systems [66, 104, 124] have been devised to mitigate these problems by either checking IPC call chains, or by monitoring the run-time communication between apps through a modification of the Android OS. Besides of them, some other approaches based on static analysis have been proposed to detect vulnerabilities. SEFA [263] is a tool that analyzes in-component, cross-component, and cross-Apps vulnerabilities, while ComDroid [78] and Woodpecker [142] focus on the detection of in-component vulnerabilities. CHEX [178] is one of the most sophisticated tools, as it analyses both in-component and cross-component communications to detect component hijacking vulnerabilities. CHEX accomplishes this task by tracking taints between externally accessible interfaces and sensitive sources or sinks. Finally, Epicc [194] goes beyond ICC vulnerability detection, as it can also be used for detecting Apps that may exploit a given vulnerability.

Permission Usage analysis

A number of studies and tools have been devoted to the analysis of the use of permissions. Barrera et al. [57] studied the permission usage patterns of third party apps by applying self-organizing maps. Stowaway [120] addressed the issue of overprivileged Apps, i.e., those Apps that request permissions more than their necessities, and don't follow the principle of least privilege, by mapping Android APIs to their related permissions. Overprivileged third party apps are also addressed by Vidas et al. [255] and PScout [52], that provide different approaches to permission mappings. Whyper [199] and AutoCog [214] are designed to bridge the gap between what user expects an App to do, and what it really does. In particular, they assist the end users to understand permissions using the available descriptions of each App. These systems automatically assess these descriptions to understand why Apps use a specific permission by leveraging on natural language processing techniques and machine learning.

Information flow analysis

As we discussed in section 2.3.3, one of the popular way to analyze Apps and detect privacy leaks is by taint analysis. Dynamic taint analysis systems are particular resilient to obfuscation attempts that leverage on the reflection mechanism, because they track all function call when they are executed, no matter whether methods are invoked through reflection or not. On the other hand, dynamic taint analysis approaches suffer from the problem of code coverage, as they can only track the calls that are executed during the analysis phase. This problem does not affect in principle static taint analysis techniques because its aim is to cover all execution paths, the main problem in this case being code obfuscation. Dynamic analysis can also be fooled by a malicious apps that can detect if it is executed for analysis purposes.

It turns out that static and dynamic analysis can provide complementary information for the analysis of the information flow. Static taint analysis is performed by SCanDroid [128], LeakMiner [276], and AndroidLeaks [132]. One of the most recent works for static taint analysis is FlowDroid [49], that has been also distributed under an open source license. FlowDroid handle callbacks invoked by the Android framework through the analysis of the App lifecycle, and it exhibits a low false alarm rate because context, flow, field and object-sensitivity are considered. FlowDroid is configured to detect all flows between a list of sources and a list of sinks inferred by their SuSi project [215], which automatically detect sources and sinks from Android OS source code by machine learning techniques. For example, if one of these tools finds a flow in an App between a source like the *getSimSerialNumber* function, which returns the serial numbers of the device SIM card, or the *EditText* function containing a password, and a sink like *sendTextMessage* function, it means that this App can send the serial number of the SIM card or the password via SMS.

Android sandboxes

In addition to Android OS instrumentation for dynamic analysis mentioned in section 2.3.3, some other approaches monitor Apps during their execution time based on virtualization [63, 239, 256]. For example, DroidScope [274] is a VM-based analysis platform which runs the whole Android platform on the Qemu emulator [29] to reconstruct both the OS and Dalvik level views of the system, such as the Memory and Register read/write APIs, the Linux System calls, and the Dalvik APIs, like the interpret Java object. Copperdroid [218] is another examples in this group, that claims high performance and portability. It automatically reconstruct the behaviors of Android malware by executing it in a Qemu environment. It monitors all system calls because the analysis is carried out by leveraging on the binder protocol, and then the analysis of the system calls is performed outside of the emulator, so that it is transparent to the malware that runs in the system. It can also extract additional behaviors by looking at the information in the manifest file as the list of broadcast Receivers. Crowdroid [69] and Paranoid [211] are other examples of dynamic analysis that run on the device causing a computational overhead estimated in 20% . In particular, Crowdroid collects system calls on the device, and sends the statistics to a centralized server. Paranoid runs a daemon on the phone to collect behaviors and then it sends the behaviors to a cloud-based server. Finally, NetworkProfiler [87] detects Android fingerprint by analyzing the real-world HTTP traffic flowing thorough the cellular provider network.

Detection of repackaged Apps

App repackaging is a kind of attack performed by first altering a pre-existing legitimate App to hide a malicious function, and then publishing the repackaged App in Google Play with a similar name. Although the digital signature changes, the users usually are not able to distinguish an official/legitimate App from malware, as the most frequent repackaged Apps are games, utilities and porn Apps. DroidDream malware is such a prominent example downloaded by 250K users in 2011 [175]. This kind of attacks can be worse if the digital signature of the malware remains the same as the original one. In 2013, The Bluebox Security research team discovered a vulnerability [64], Android Master Key Exploit, in the Android security model which was patched by Google in February 2013. The vulnerability allowed the attackers to inject malicious code into legitimate apps without invalidating the digital signature, completely unnoticed by the app store, the phone, or the end user. This serious Android vulnerability allowed an attacker to hide code within a legitimate App, repackage the App, and then use the existing permissions to perform sensitive functions through those apps.

To defend against this threat, DroidMOSS [289], DNADroid [84], and PiggyApp [288] were developed for detecting repackaged Apps in the Android App market. DroidMOSS leveraged on opcodes in the Dalvik byte code, while DNADroid used program dependency graphs between methods, PiggyApp leveraged on a number of information as the requested permissions, the Android API calls used, the involved intent types, the use

Table 2.5: Comparison between App analysis techniques

Name	Type	ICC	NC	Analyzed Data	Auxiliary Tool
Information flow analysis					
SCanDroid [128]	S	✓	×	FCG, ACO	WALA
FlowDroid [49]	S	×	×	FCG, LFC, ACO	Soot, Dexpler, Heros
DidFail [162]	S	✓	×	CFG, FCG, LFC, ACO	FlowDroid, Epicc
IccTA [167]	S	✓	×	CFG, FCG, LFC, ACO	FlowDroid, Epicc
Vulnerability analysis					
Woodpecker [142]	S	✓	×	CFG	baksmali, Soot, WALA
CHEX [178]	S	✓	×	CFG	DexLib
ComDroid [78]	S	✓	×	CFG	Dedexer
Epicc [194]	S	✓	×	CFG, FCG	Dare, Soot, Heros
SEFA [263]	S	✓	×	PER, API, FCG	—
Fratantonio et al.'14 [208]	S	×	×	CFG	AndroGaurd
Repackaged App Detection					
PiggyApp [288]	S	✓	×	PER, API, AUT	baksmali
DNADroid [84]	S	×	×	PDG	baksmali
DroidMOSS [289]	S	×	×	OPC	baksmali
Malware Detection					
DroidMat [262]	S	✓	×	PER, ACO, API	apktool
Drebin [48]	S	✓	×	PER, ACO, HCO, API, STR	—
DroidMiner [275]	S	✓	×	FCG, CFG, API	—
DroidSIFT [280]	S	✓	×	ADG	Soot, Epicc
Crowdroid [69]	D	✓	✓	SYC	Strace
Anomaly Detection					
pBMDS [268]	D	✓	✓	SYC	Qtopia
Andromaly [232]	D	×	×	SMD, NST	—
STREAM [41]	D	×	×	SMD	Emulator, Monkey
Shabtai et al.'14 [233]	D	✓	✓	NST	—
App monitoring sandbox					
CopperDroid [218]	D	✓	✓	SYC	QEMU, AndroGaurd
DroidScope [274]	D	✓	✓	API, OPC	QEMU

-Information extracted from Manifest file

ACO: App components, HCO: Hardware Components, PER: Permissions

-Information extracted from disassembled code

ADG: API dependency Graph, API: App Programming Interface, CFG: Control Flow Graph

FCG: Function Call Graph, OPC: Opcode of Instructions, PDG: Program dependency Graph

STR: Strings, LFC: Lifecycles

-Information extracted from META-INF subdirectory

AUT: Authorship information

-Information extracted during run-time

NST: Network Statistics, SYC: System calls, SMD: System metadata of native code or external classes, as well as the authorship information to discriminate repackaged Apps from the original App in the Google market or third-party markets. DroidRanger [296] was a further approach that combined both static and dynamic analysis to detect malicious Apps in the existing Android marketplaces. Vidas et al. [253] showed that some alternative markets distribute repackaged Apps containing malware. Their proposed approach, named Appintegrity, mitigated this situation with a simple verification protocol. AppInk [287], on the other hand, prevented the repackaging of legitimate Apps by including a transparently-embedded watermark when generating the APK from the source code. Finally, DIVILAR is an approach based on a virtualization based protection scheme to enable self-defense against App repackaging [286].

Anomaly detection

Anomaly-based malware detection systems refers to algorithms designed to detect Android malware based on heuristic rules rather than signatures. Anomaly-based detection is usually carried out in two phases: the training (or learning) phase, and the detection phase. The main advantage of anomaly-based detection systems is their ability to generalize from known malware samples, and thus detect new malware samples. On the other hand, their drawback is the potential generation of high false alarm rates. In order to maximize the detection rate, while keeping low the rate of false alarms, one of the most important issues is the extraction and selection of effective features that allow discriminating malware and benign apps.

Teufl et al. [249] proposed a learning-based malware detection system based on the metadata associated to each app within the Google play market, such as the permissions, the ratings, and information about the developer. DroidMat [262] is one of the first static analysis systems for Android malware detection based on the analysis of App packages by machine learning techniques. It considers static information such as permissions, components, Intents and API calls for characterizing the behavior of Android Apps. Then, it applies the K-means clustering algorithm to categorize Android malware. Finally, it uses the k-NN classification algorithm to classify the App as benign or malicious. Some approaches [75, 203, 228] use machine learning techniques to automatically classify apps as potentially malicious based on the permissions. In addition to permissions, Drebin [48], DroidMiner [275] and DroidSIFT [280] extract more information through static analysis such as, APIs usage, strings, the control flow graph, the API dependency graph, and use these information as a set of features to distinguish malware from benign Apps.

Static analysis approaches offer a partial view on the behavior of the system, as not all the behaviors observed at runtime can be captured. Accordingly, a number of techniques, which are just few, have been proposed that base of their analysis on features extracted at runtime [99, 268, 278], thus the name is dynamic analysis. For example, Stream [41] is a distributed App analysis tool that leverages on system metadata from battery, binder, memory, network, and permissions to perform malware detection, while Andromaly and some other similar techniques [186, 232, 233] perform Android malware detection by the analysis of network traffic generated by the Apps.

2.4.5 App layer access control

The Inline Reference Monitor (IRM) [112] is a mechanism that has been proposed to enable access control by adding a reference monitor to the App. With respect to the access control mechanisms in the kernel layer, that we discussed in section 2.3.3, the controls enforced by IRM don't require context switches in the kernel, so it has a lower performance overhead. Two examples of such solutions specifically developed for Android systems are AppGuard [56], and Dr. Android and Mr. Hide [155]. Aurasium [271] is another IRM solution that first repackage each App with an injected native library. The repackaged App can then be installed on a user's phone, so that any defined policies can be enforced at runtime without altering the original App functionalities. This mechanism can enable monitoring Internet connections, IPC binder communications, and file system manipulations. The main deficiency of IRM solutions is that a malicious App can detect the presence of an IRM, and can then bypass the system by native code or Java reflection.

2.4.6 Protection against App analysis

One common way to protect Apps against static analysis is through *obfuscation* mechanisms. The term obfuscation refers to a number of techniques that makes App and its textual data hard to understand. Software developers sometimes employ obfuscation techniques because they want to protect their programs against reverse-engineering attempts. ProGuard [42] is a built-in tool in the Android SDK that obfuscates and optimizes the code by removing unused code, renaming classes, fields, and methods. This free tool has a commercials companion, DexGuard [22], with the additional ability of performing more sophisticated obfuscations than the ones included in ProGuard, such as native-code and class encryption, emulator detection, and the use of reflection mechanisms for function calls. Malware coders have used obfuscation techniques to make their Apps more difficult to analyze, and thus evade detection. DroidChameleon [217] showed that a large fraction of commercial Android anti-malware products can be evaded through the use of *trivial* obfuscation techniques. More recently [179] showed that current versions of anti-malware products for Android are robust against *trivial* obfuscation mechanisms, even if they are still vulnerable when sophisticated techniques are employed.

In addition to techniques aimed at mislead or evade detection by static analysis tools, [207, 254] explored the possibility to fingerprint Android sandboxes to evade dynamic analysis tools. They found that all Android sandboxes are susceptible to evasion as any Android App can check for the presence of emulators by some heuristics. To defend against these evasion attempts, Morpheus [157] has been proposed as a framework that systematically discovers the execution of such heuristics to prevent malicious Apps from bypassing emulator-based analysis. Recently, it has been shown that static analysis can be completely circumvented, while dynamic analysis can be somewhat circumvented, by hiding the malicious code through *dynamic code loading* techniques. This technique works as follows. Apps can load a piece of code (e.g. downloaded from the Internet) dynamically at runtime without requesting any permissions. There are different approaches such as loading JAR files, DEX files, Linux shared objects (Native code), or using code from other apps. The only verification mechanisms that are enforced on the external code loaded by the Application are the App's permissions. Consequently, malicious apps can use this technique to circumvent offline analysis tools as the *Verify Apps* tool used before an App is published in the Google Play market. On the other hand, benign apps can use code-loading techniques for downloading updates, as well as advertisements, so potentially causing dangerous vulnerabilities that allows code injection threats by hijacking the HTTP connection [62]. One of the solutions proposed so far in the literature [208] consists in the verification and check against a whitelist of the piece of code that is loaded, before it is executed.

Chapter 3

Clustering Android Malware Families by Http Traffic

3.1 Introduction

A botnet is a network of compromised machines (bots) commanded and controlled by a bot master for massive attacks such as dispatching unsolicited emails (SPAM), launching Distributed Denial of Service (DDoS) attacks, and performing information theft. Botnets leverage on different approaches such as encrypted HTTP protocol, fast-flux, and domain-flux techniques to be resilient against detection. Botnets may rely on 2 types of command and control (C&C) channels: (i) centralized C&C such as IRC and HTTP; (ii) distributed C&C such as P2P. C&C traffic is hard to identify and to be blocked either at the network level (e.g., by setting appropriate rules on a firewall) or at the DNS level (e.g., by domain blacklisting).

As the source of the user-generated network traffic is moving from desktop computers to mobile devices, mobile malware have become a serious concern that target in particular the Android platform [238]. Android botnets [191] are malware families that take control of Android devices in the same way as malware that are designed to set up a botnet of desktop computers. Commands to mobile bots are sent through different channels such as HTTP, SMS, and the Google Cloud Messaging (GCM) service. Although the Android system itself and mobile anti malware products introduced many security policies and techniques to protect Android devices against malware, mobile botnets are still on the rise.

As large numbers of new mobile malware samples are collected on a daily basis, new techniques are needed for a fast and accurate assessment of the *family* the malware belongs to. We focus our analysis on malware samples that send and receive data using the HTTP channel. We chose HTTP for our study as 70% of the generated network traffic by Android apps is spread through this protocol [87]. In addition, most of the web-based traffic generated by Android apps does not use the HTTPS protocol [259]. In particular, more than 99% of Android botnets use the HTTP-based web traffic to receive bot commands from their C&C servers.

In this chapter we show that mobile malware can be effectively clustered and detected on the basis of statistics calculated on the HTTP traffic that is generated by the applications. To do so, we leveraged on a previous work [205] that proposed a technique to cluster and detect malware for desktop architectures. Results show that not only the same rationale is still valid for mobile devices, but also that a simpler system can be used when dealing with mobile malware.

In summary, this chapter aims to answer the following research questions:

RQ1 Can we use the HTTP network traffic generated by Android apps to detect malware families?

RQ2 Which features extracted from the HTTP traffic are effective for clustering and detecting Android malware?

RQ3 How well do the performances on Android malware compare to the ones on desktop malware?

Accordingly to such questions, the contributions of the chapter are the following:

- We show that the analysis of the HTTP traffic is prominent for the detection of Android malware.

- We propose a malware detection system that is both efficient and effective, as it leverages on seven statistical features that allow for reliably clustering Android malware into families. From such cluster we extract signatures with which it is possible to precisely detect malware belonging to the clustered families.

- The overall performances of the system for mobile malware clustering and detection are better than the ones related to a similar system developed by some of the authors for traditional malware [205]. The reason behind this behavior can be related to the smaller variability in the statistics of the HTTP traffic, as HTTP communications among apps are generally more limited when compared to desktop ones. In addition, mobile botnets have to control less functionalities and applications compared to Desktop ones. In this way, the traffic generated by malware samples belonging to the same family can be more easily separated from the traffic generated either by benign applications or malicious applications belonging to different families.

The rest of this chapter is organized as follows:
The method used for the experimental evaluation is presented in Section 5.3. Section 3.3 shows the results of the experiments carried out on a significant dataset of Android malware samples. Section 3.4 provides an overview of the closest state of the art approached to ours, as well as comparing them with our work; Conclusions are drawn in Section 3.5.

3.2 Proposed System

The architecture of the proposed system is depicted in Figure 3.1. The detail of data gathering step expose in §3.3.1 while the clustering and the signature generation step is discussed as follows.

We relied on an algorithm that has been proposed by some of the authors for clustering malware that target traditional desktop systems [205], and tested if the proposed scheme

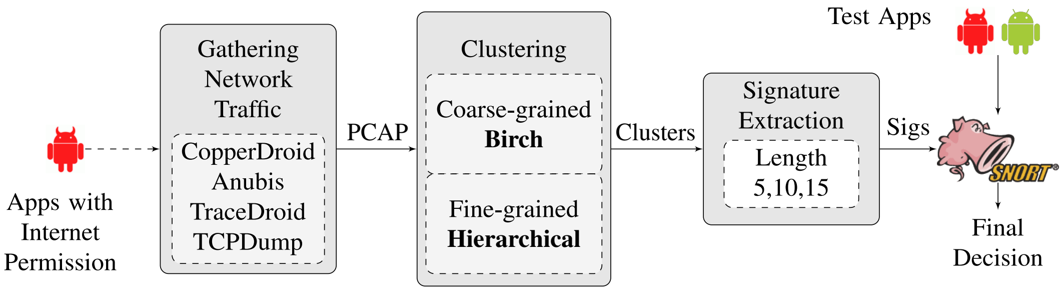


Figure 3.1: Overview of our approach.

was still valid for clustering Android malware on the basis of the HTTP traffic that they generated. The algorithm adopts a multi-step clustering procedure to define the clusters and generate the signatures for malware detection. The multi-step procedure was proposed to speed-up the process, by first using statistical traffic features to perform a coarse- grained clustering, and then by employing a set of structural features (i.e., features that take into account the content of each HTTP connection) to perform a fine-grained clustering. Both the coarse-grained and fine-grained clustering procedures are carried out by resorting to hierarchical clustering techniques, where data is aggregated in nested clusters and the clustering process terminates when further aggregation merges two distant clusters. In the following we will briefly recall these steps:

1. ***Coarse-grained Clustering***: to perform this step, the BIRCH algorithm is employed [281]. The main goal of BIRCH is to perform approximate clustering of arbitrarily large datasets with a guaranteed memory bound and with I/O access costs that grow linearly with the size of the dataset. Whenever the clustering process approaches the preset memory limit, the clustering algorithm will further compress the dataset, thus producing a less fine-grained representation of the data and thus resulting in fewer, larger clusters. The term *coarse grained* refers to the use of simple statistical features extracted from the HTTP traffic to characterize the connections. The seven features that have been used are the following:

   f_1 : the total number of HTTP requests

f_2 : the number of GET requests

f_3 : the number of POST requests

f_4 : the average length of the URLs

f_5 : the average number of parameters in the request

f_6 : the average amount of data sent by POST requests and

f_7 : the average length of the response

The size of a cluster can be measured by its *radius*, whose value can be limited to avoid generating too large clusters that might incorrectly group the connections. After this process, the clusters that are obtained will contain HTTP connections featuring similar statistical features. However, such connections might be related to different malware families. For this reason, each cluster needs to be further refined through a *fine-grained* clustering process, in order to further split the coarse-grained clusters that might contain different families.

2. ***Fine-grained Clustering***: To perform this step, a single-linkage hierarchical clustering algorithm is used and the distance between HTTP requests is computed according to the four parameters:

   f_1 : the request method used

   f_2 : the page

   f_3 : the set of parameters names

   f_4 : and the set of parameters values

   The reader that is interested in a detailed description of the distance computation employed at this step could check [206].

3. ***Signature Generation***: Whereas clustering allows for grouping together malware samples belonging to the same *family*, further information can be extracted by the samples in the same cluster, leading to the generation of a signature that can be used for detection. To this end, the *Token-Subsequence* algorithm described in [189] can be used to extract a signature from each group of malware. These

signatures are then used by a network IDS to perform the detection of malicious traffic generated by malware samples.

3.3 Experiments

This section represents the main contribution of our work, as we aim to assess if the technique proposed for traditional desktop malware can be used to effectively cluster Android malware. In particular, our experiments had three main goals:

- verifying if the HTTP traffic generated by Android malware can be used to reveal the family they belong to;
- assessing which of the features that can be extracted from HTTP traffic are effective for malware clustering;
- checking if the result of the clustering process allows for extracting malware signatures that could be used by a NIDS.

In the following, we first describe the dataset used in the experiments and we present the measures used to evaluate the results. Then, we report and discuss all the experimental results that we attained.

3.3.1 Dataset

We gathered a large number of malicious Android applications to evaluate the effectiveness of the clustering procedure, and a large number of benign Android applications to evaluate the effectiveness of the signatures in terms of false positive rate. For the malicious samples, we focused on Android malware families that were related to botnets. We also analyzed malware families that delivered information to a remote server through HTTP.

The samples were gathered from Malgenome [292], Contagio [201], Drebin [48] and VirusTotal [35]. There were many malware families that interacted with their C&C

servers by HTTP, SMS messages, emails, etc. As the majority of malware families use the HTTP protocol, we just considered the families that employ HTTP for their C&C channel. For each sample, we extracted the HTTP information by using CopperDroid [14], or by employing either Anubis [21] or TraceDroid [33] for the cases in which CopperDroid could not generate network traffic. To retain the samples that actually generated network traffic, we removed the ones that did not produce any valid HTTP communication due to the unavailability of their C&C servers.

To avoid inaccuracies in assigning each malware sample to a family, we developed a tool[1] that automatically scans each sample using Virustotal, and creates a naming convention based on the outputs of different anti-malware products. The list of the considered malware families is shown in Table 3.1.

[1] https://github.com/ManSoSec/Auto-Malware-Labeling

Malware family	# Samples	Malware family	# Samples
AndroRat	11	Fjcon	106
AVPass	1	Geinimi	24
BackFlash	2	GoldenEagle	2
BadNews	2	Lien	6
BaseBridge	112	NickiSpy	2
BgServ	45	Obad	1
Chulli	2	Plankton	119
DroidKungFu	86	RootSmart	25
Extension	69	Skullkey	1
FakeAngry	151	SMSpacem	5
FakePlay	8	Tracer	13
FakeTimer	13		
		Total	**806**

Table 3.1: Malware families used for the experimental assessment of the effectiveness of the HTTP clustering procedure

Benign Dataset	Number of requests
Web Browser	1102237
Android Apps	1037555
Total	**2139792**

Table 3.2: Number of benign requests generated by browsing web sites, and by Android applications

There are two columns in Table 3.1. The Malware family column refers to the name of the malware variant, and the related number shows the considered malware samples for that variant.

In order to evaluate the false positive rate, we also collected a dataset of legitimate traffic. We collected over 2×10^6 HTTP requests from October 2014 to December 2014. Part of this traffic was achieved by sniffing the HTTP requests generated by crawling 173 most visited web sites (without considering search engines) in an Android emulator. The other part of the benign dataset contains HTTP traffic generated by Android applications. To generate such traffic, we performed two steps: (i) we collected applications that feature the permission *android.permission.INTERNET* by crawling[2] Google Play. We also obtained the thirty top free applications in Google Play under different categories such as Comics, Communication, News and Magazines, Shopping, Social, Sports, and Travel; (ii) we emulated the execution of the Android applications by simulating a real user behavior through the Android testing framework named Uiautomator [34]. Uiautomator lets users test the application user interface (UI) efficiently, and we exploited it to automatically interact with all of the elements in the applications' layouts to generate HTTP traffic. During the interaction, the network traffic is captured by the aid of the tcpdump tool, which collected several Gigabytes of benign HTTP traffic. As shown in Table 3.2, both the web browser procedure and the collection of HTTP traffic from benign applications produced around one million requests.

We extracted the seven statistical features from HTTP requests and responses by Jnetpcap [26] Java library. Because the values of the features are in different ranges, we performed feature normalization by re-scaling feature values according to the following equation:

$$x_i^* = \frac{x_i - min(x_i)}{max(x_i) - min(x_i)} \tag{3.1}$$

where the min and max are computed across the whole dataset.

3.3.2 Evaluation of the proposed system

To evaluate the performances of the system, the following measures have been used:

[2] https://github.com/egirault/googleplay-api

(A) `Cohesion:`

The cohesion of a cluster C_i measures the average similarity between two samples in the cluster with a value between zero and one. It will be maximum (one) when the malware samples in a cluster belong to the same family, i.e., when they share the same label. The value zero, on the other hand, indicates that the malware samples are from different families. Whereas the HTTP traffic is similar according to the features employed, the labels of the samples are different. The average value has been computed only for those clusters that contain at least two malware samples. In fact, clusters containing just one malware sample have a cohesion equal to one by definition, and including them would mistakenly bias the final result.

(B) `Separation:`

The separation between two clusters C_i and C_j measures the average family label distance between malware belonging to C_i and malware belonging to C_j. This gives us an indication about whether the malware samples in two different clusters belong to different malware families or not. In order to understand how much the whole of clusters are well separated, instead of calculating the average of the separation between clusters, we calculate the percentage of the number of clusters that have a separation above a threshold. The detailed definitions of the measures of cohesion and separation can be found in [206].

(C) `Detection rate:`

The detection rate measures the percentage of malware that is detected by relying on the signatures extracted by the clusters. The detection rate is calculated for each value of the radius parameters that controls the number of generated clusters in the BIRCH algorithm. The extracted signatures are then included in Snort, [31] which is used to process network traces produced by malware samples and legitimate apps. Then, the alerts are collected and the detection rate is calculated as follows:

$$D.R.(\%) = \frac{N_{malware}}{N} * 100 \tag{3.2}$$

where $N_{malware}$ is the number of samples for which Snort generates at least one alert, and N is the total number of samples.

(D) `False positive rate:`

The false positive measures the percentages of false alerts that the IDS outputs according to the signatures extracted from the clusters, and it is computed according to the following formula:

$$F.P.(\%) = \frac{N_{alerts}}{N_{requests}} * 100 \tag{3.3}$$

where N_{alerts} is the sum of all alerts and $N_{requests}$ is the number of all the requests.

3.3.3 Experimental Results and Discussion

Cohesion vs. Separation

We carried out the experiments by running the complete two-step clustering algorithm. First, we run BIRCH on the statistical features, and we applied the single-linkage hierarchical algorithm to refine the clustering result of the first step. Then, we compared these results with the ones obtained by running only the BIRCH algorithm on the statistical features. The metrics used to compare the results are Cohesion and Separation. We computed these two metrics for 16 values of the R parameter, i.e., the *radius* that controls the number of clusters generated by BIRCH in the range from 10^{-7} to 10^{0}.

Surprisingly, we noticed that using the two-step algorithm leads to the same Cohesion and Separation values that are obtained when using BIRCH stand-alone. Figure 3.2 shows the values of the average Cohesion of all clusters for different values of radius R used in the experiments. Cohesion exhibits only small variations with different values of the radius. In particular, the average Cohesion is around 0.95 for values of the radius lower than 10^{-3}, and it slightly decreases in the interval between 10^{-3} and 5×10^{-1}. From 0.88, the average Cohesion increases again. Achieving high Cohesion means that the malware inside each cluster are very similar to each other and therefore they are properly clustered in the same family. However, we will see that this is not enough to produce reliable malware signatures. Figure 3.3 shows the values of the Separation index. The percentage of pairs of clusters with a separation index higher than 0.1 ranges from 84.01% to 96.66%. Attaining high Separation values means that the clusters are better separated. By examining the two figures, it turns out that the best trade-off between Cohesion and Separation is for values of the radius between 0.5 and 1.

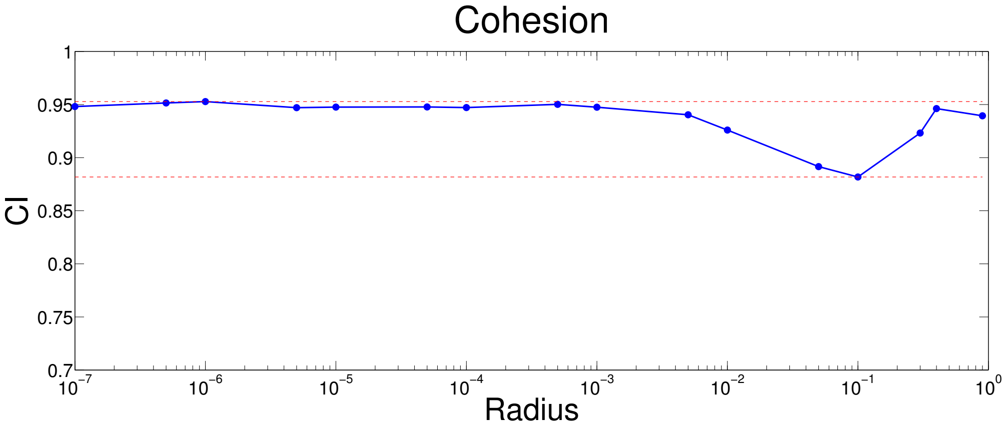


Figure 3.2: The average value of the cohesion indexes CI.

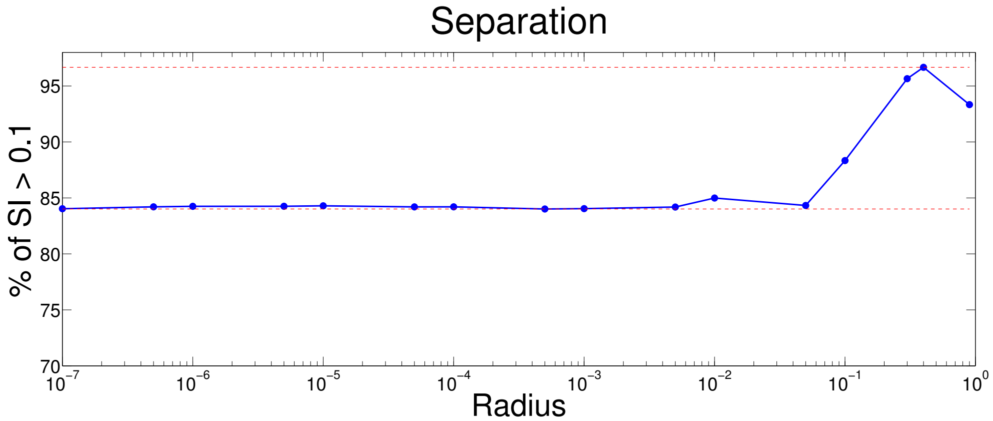


Figure 3.3: The percentage of pair of clusters with a separation index SI higher than 0.1.

Detection Rate vs. False Positive Rate

To compute the detection rate and the false positive rate, we extracted the malware signatures (which are a part of the request URL) from the clusters. In the experiments, we extracted different signatures for three different values of the minimum (Min) length of the signatures, i.e., 5, 10, and 15. Some examples are shown below :

Sig1: *content:"POST"; distance:0; nocase; content:"/ad"; distance:0; nocase;*
The above signature has length 3, which is not in none of Min 5, 10, and 15 categories.

Sig2: *content:"POST"; distance:0; nocase; content:"/aar.do"; distance:0; nocase;*
The above signature has length 7, which is in Min 5 category but not in 10 and 15 categories.

Sig3: *content:"POST"; distance:0; nocase; content:"/api/proxy"; distance:0; nocase;*
The above signature has length 10, which is in Min 5 and 10 categories but not in the Min 15 category.

Sig4: *content:"GET"; distance:0; nocase; content:"/adv/d?t=135782568"; distance:0; nocase;*
The above signature has length 18, which is in all of Min 5, 10, and 15 categories.

The detection rate calculated for the three different signature lengths and for the values of the radius reported in the previous subsection are shown in Figure 3.4. The values of the detection rate are quite high, and allow for precisely detecting the network traffic generated by malicious applications. The false positive rate accounts for the specificity of the signatures, and it is computed to evaluate the fraction of benign HTTP requests that match with the signatures. The results are shown in Figure 3.6. Both the detection and false positive rates do not improve if the two step clustering is employed instead of the stand-alone BIRCH (see Figure 3.5 and Figure 3.7). Although achieving lower Cohesion and higher number of rules in the range of $10^{-}3$ and 10^{0} could lead to producing more signatures with less integrity, it could significantly worsen the false positive rate.

The appropriate set of signatures are those that allow attaining a high detection rate and low false positives. To this end, we observe that the best values for the detection rate (very close to 100%) were obtained by signatures with minimum length equal to five for all values of the radius. However, if we take into account the corresponding values for the false positive rate, it turns out that the best trade-off is reached when the radius = 0.9, which allows to attain a false positive rate of around 7%.

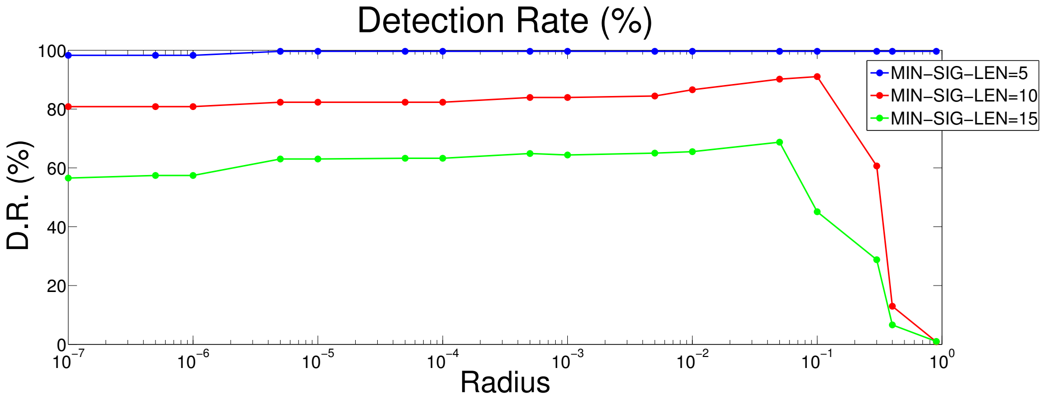


Figure 3.4: The percentage of detection rate D.R. (%) obtained with different set of signatures and different values of the radius, by only doing coarse-grained clustering.

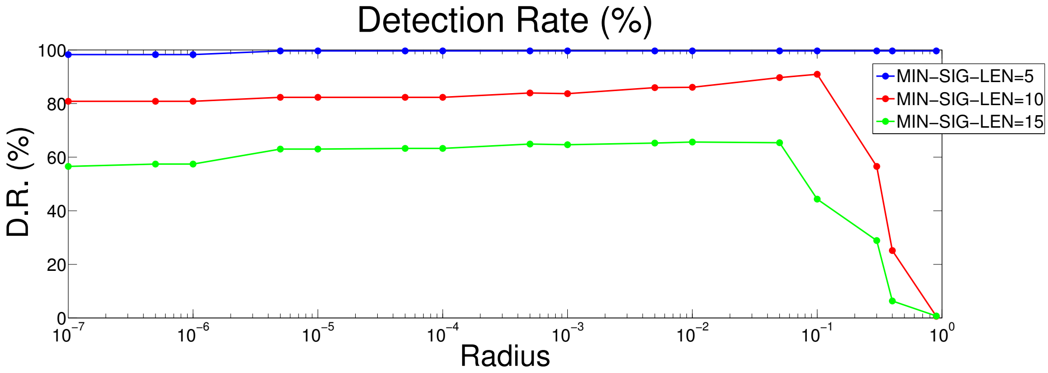


Figure 3.5: The percentage of detection rate D.R. (%) obtained with different set of signatures and different values of the radius, by doing fine-grained clustering in addition to coarse-grained clustering.

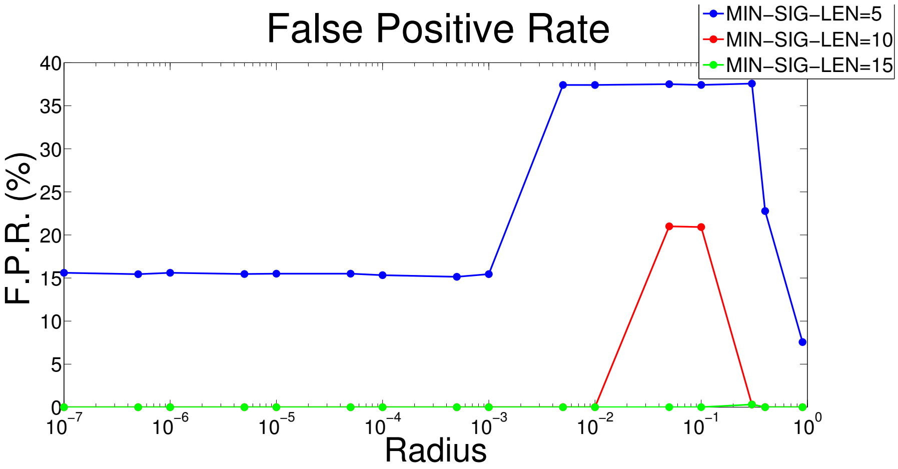


Figure 3.6: The percentage of false positive rate F.P.R. (%) obtained with different set of signatures and different values of the radius, by only doing coarse-grained clustering.

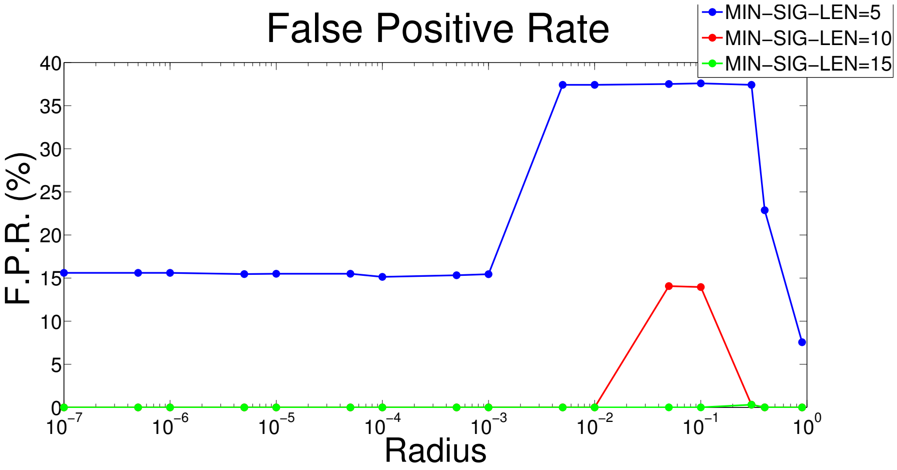


Figure 3.7: The percentage of false positive rate F.P.R. (%) obtained with different set of signatures and different values of the radius, by doing fine-grained clustering in addition to coarse-grained clustering.

Number of Clusters and Number of Signatures

Although the efficiency of the detection system is important, the number of clusters (Figure 3.8) and signatures (Figure 3.9) need to be evaluated. The number of clusters is correlated to the value of the radius. If the radius is large, the number of clusters is small, meaning that each cluster is likely to contain malware from different families. In our experiments, the number of clusters for a radius value = 10^{-7} is 501, whereas the number of clusters is 5 for radius = 0.9. As we mentioned, the Snort rules are based on the signatures generated from the clusters, and they are correlated with the number of signatures. A large number of clusters may translate into a redundant number of signatures, as malware belonging to the same family may be grouped in different clusters. Consequently, even if the detection rate does not vary, the false positive rate and the IDS speed exhibit lower performances.

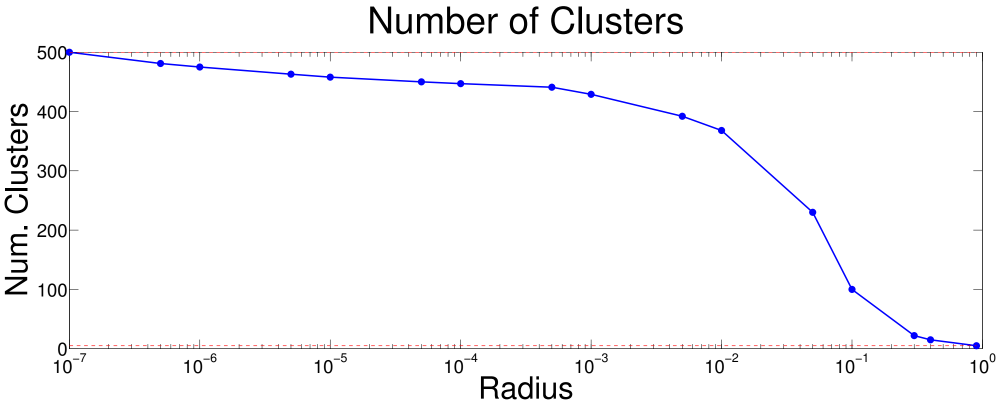


Figure 3.8: The total number of clusters (Num. clusters) for different values of the radius.

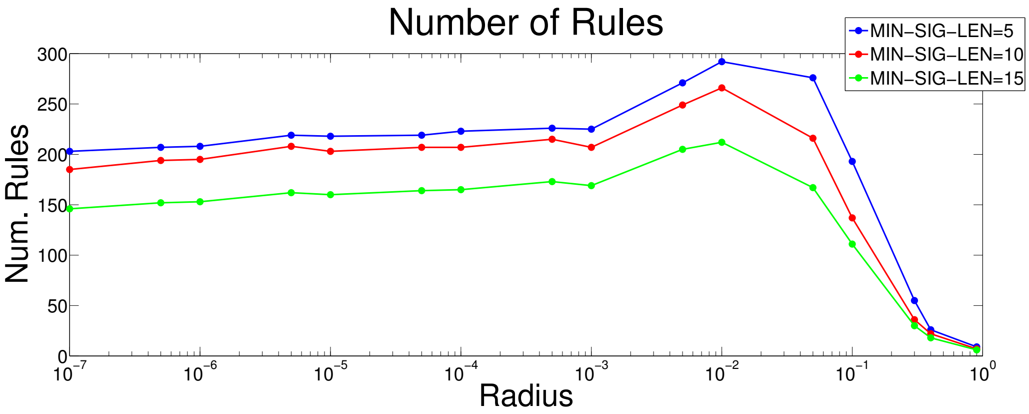


Figure 3.9: The total number of the Snort rules (Num. rules) obtained from the signatures for different radius values.

3.3.4 Comparisons with HTTP based clustering for traditional desktop malware

As Android has been largely adopted only recently compared to traditional desktop systems, the number of available Android malware samples that generate malicious HTTP traffic are fewer than the analogous desktop malware samples. In general, compared to the work on traditional desktop malware [205], clustering Android malware samples by HTTP traffic traces shows that: (i) the value of cohesion is higher, (ii) the value of separation is lower, (iii) the detection rate is higher, (iv) and the false positive rate is lower. Thus, grouping malware samples according to the generated HTTP traffic they produce is effective not only to detect malware families, but also to produce effective malware signatures. In order to show the basic motivation behind this behavior, we computed the mean and the variance of each statistical feature for malware samples belonging to both platforms. We show this comparison in Table 3.3.

Windows malware exhibit a higher mean value on features such as the number of Get requests, and the length of sent and response data. Conversely, the number of Post requests and parameters, and the length of URL are larger for the Android malware. The limited capability of mobile devices compared to desktop PCs can be

Platform	Get	Post	Url	Param	Sent	Response
Windows	19.1(183.1)	1.1(10.3)	51.9(58.0)	1.4(3.3)	84.5(201.5)	81E+4(88E+5)
Android	3.8(6.1)	2.8(3.3)	87.9(98)	4.4(6.5)	73.7(97.0)	88E+2(15E+4)

Table 3.3: Comparison on Average (Standard deviation) for each statistical feature

the reason for the low interaction of mobile malware with their C&C servers. In other words, desktop applications tend to produce multiple HTTP requests to perform an action, and Android apps tend to produce one long HTTP request containing all the information. Another possibility is that mobile botnets have to control less functionalities and applications compared to Desktop ones. For example, the higher value of the average Get requests for Desktop malware shows that such malware resort to more computational resources to generate as many requests as possible. Consequently, they do not need to create long URLs with different parameters. Conversely, as mobile devices have limited computational capabilities, malicious applications are developed to keep the number of requests as small as possible. Thus, they are forced to enrich their requests by using a larger number of parameters. In addition, the analysis of the standard deviation values allows for a better understanding of the diversity of requests. As an example, the diversity of the number of requests and of the amount of transferred data is much higher in Desktop malware compared to Android malware. This examples provide reasons for the effectiveness of the clustering by just using the statistical traffic features, and for the higher performances in malware detection for mobile devices compared to desktop systems.

3.4 Comparison with Related Works

There are some proposed security mechanisms to analyze malicious Android app by observing the network traffic they generate. In the following, we will briefly review some of the mechanisms.

Andromaly [232] is a malware detection system that employs machine learning and leverages information such as the CPU usage, active processes, and the amount of transferred data through the network. In another work [233] by the same authors, the analysis was focused on the network traffic generated by Android applications. The main goal of this system was the identification of malicious attacks perpetrated by means of repackaging. The authors showed that applications could be subdivided into a number of categories according to the statistics of their transferred data. They also concluded that deviations from specific normal behaviors could be classified as malicious activity.

Narudin et al. [186] proposed another detection approach based on both TCP and HTTP protocols. They considered network traffic features such as some basic information from the TCP header (e.g., the frame length), content based features such as the number of HTTP requests, and time-/connection- based features such as the number of frames in a specific time interval or connection. Another nearly similar detection approach [47] was proposed by using a classification system, which is fed just with some time-/connection-based features, and was tested on a small dataset containing 43 malware samples.

NetworkProfiler [87] is another approach that performs application analysis based on the HTTP header. They generated fingerprints from the network usage of each app, and they were able to use them to detect malicious activities by inspecting the traffic logs produced by a network provider. Subsequently, the same authors resorted to network traces to identify Android applications that use in-app advertisements [250]. Zarras et al. [279] proposed a system that analyzes, among the others, the sequence of headers in HTTP communications to detect malicious traffic generated by botnets. To this end, they extract HTTP traffic generated from both desktop and mobile applications.

Apart from identifying coarse-grain behaviours such as the presence of maliciousness in a network traffic flow, extracting finer-grain information from the device communications can be interesting for an adversary. Conti et al. [81] designed a system based on network flows analysis and machine learning techniques to identify user actions such as sending an email or posting a message on a friend's wall in online social networks. Wu et al. [264] proposed an approach based on extracting the characteristics of the app from the HTTP traffic to detect repackaged applications on the Android markets.

Approach	Analyzed protocol		Purpose
	TCP	HTTP	
Andromaly [232]	✓		Malware detection
NetworkProfiler [87]		✓	Android fingerprinting
Tongaonkar et al. [250]		✓	Identification of apps with ads
Shabtai et al. [233]	✓		Malware and Repackaging detection
Narudin et al. [186]	✓	✓	Malware detection
Arora et al. [47]	✓		Malware detection
Conti et al. [81]	✓		Identification of user actions
Wu et al. [264]		✓	Detect repackaged apps
The method in this work		✓	Malware detection

*All the systems use machine learning techniques

Table 3.4: Comparison of different network analysis techniques for Android applications

Table 3.4 provides a summary of the aforementioned network-based analysis approaches, and compares them to the characteristics of the method that we employ in this work. Our method is the first approach that extracts a few statistical features just from the traffic HTTP header for the task of Android malware detection.

3.5 Conclusions

In this work, we performed an analysis of Android botnets that employ HTTP traffic for their communications. By clustering the generated network traffic of different Android malware with the usage of an algorithm originally developed for grouping desktop malware, we showed that the samples belonging to the same malware family have similar HTTP traffic statistics. Moreover, a small number of signatures can be extracted from the clusters, allowing to achieve a good trade-off between the detection rate and the false positive rate. The possible reason of this behavior can be related to the higher uniformity of the HTTP traffic generated by Android bot clients compared to the traffic generated by Desktop bot clients.

Chapter 4

Detecting Misuse of Google Cloud Messaging in Android Badware

4.1 Introduction

Mobile applications are often developed as client interfaces for accessing services provided by remote servers. In this setting, one of the challenges for developers is to timely notify mobile applications, i.e., the clients, on any event that updates the status of the application; e.g., messaging applications like WhatsApp need to notify their clients when they receive a new message. It is clearly computationally convenient that an application is notified only when new information is available on its server (i.e., through a *push* notification), rather than frequently checking if there is a new message (i.e., using a *pull* technique).

One of the most used services that allows implementing push notifications for Android applications is Google Cloud Messaging (GCM). Thanks to its efficiency and simplicity, GCM has also attracted the attentions of attackers. In fact, there are preliminary evidences of the use of this library in several unwanted applications like adware and bots, which we generically refer to here as *badware*, based on the ENISA threat taxonomy [1].

One possible case of GCM misuse is when it is transitively used in adware as many advertisement libraries (adlibrary) use GCM. There is a belief that this type of software is not exactly badware, and the boundary between adware and free benign applications using built-in adlibraries is rather blurred than clearly defined [48]. However, applications displaying ads are often undesired, because they drain battery life, consume unnecessary bandwidth, and can slow down the app [144, 212]. In addition, they may also exhibit sophisticated malicious behaviors like rooting the device.[1] In addition to being used in adware, GCM exhibits a number of desirable properties for attackers, rather than pull services like HTTP, to be engaged as a command and control (C&C) channel. The potential misuse of GCM in botnets was reported by the security community in 2012 [285], but the first real variant of botnet exploiting GCM for C&C was reported by Kaspersky in 2013 [160]. Less than a year later, another report by AndroTotal discussed the interest of attackers to exploit GCM channels in a malicious manner [43]. We discuss the GCM mechanism in more detail in Section 4.2, along with examples of how it can be exploited in malicious Android applications.

The only existing way to thwart GCM-based attacks is blocking the app's GCM registration ID at the GCM servers. However, this requires one to first detect the badware channel, and no solution to assess the degree of suspiciousness of GCM channels has been developed yet. One possibility could be to monitor network traffic of GCM channels, and detect anomalous behaviors. Although such a solution may enable detection of 0-day (i.e., never-before-seen) botnet channels while operating at the server side, GCM messages might be encrypted to circumvent tracking and detection. This motivates our proposal of a client-side solution, presented in Section 4.3, in which we model the functionality of GCM regardless of the content of messages, to be effective also against message encryption. In Section 4.4, we empirically show that characterizing GCM services is useful to achieve a more accurate detection of bot clients, as well as unwanted adware. Our results show that the detection rate can be increased up to 2.4%, while the false positive rate can be decreased up to 1.9%.

[1] http://www.androidauthority.com/new-android-adware-nearly-impossible-to-remove-654197/

To summarize, this chapter provides the following two main contributions.

(i) We build a model of GCM communications to evaluate the extent to which this popular mechanism is misused in Android applications. To this end, we provide a flow analysis for GCM services to be able to automatically detect flows originated from GCM entrypoints.

(ii) We show how the extracted flows from GCM services can help one to more effectively detect badware using GCM, where GCM contributes in the realization of the malicious activities. Our approach for badware detection is based on machine learning and, in particular, on a multiple classifier system (MCS) architecture.

We conclude this work by discussing the closest related work (Section 4.6) and future research directions (Section 4.7).

4.2 Background

To better understand the whole GCM mechanism and how this service can be misused by an attacker, in this section we first discuss how GCM works, and then report an example of a dissected GCM badware.

4.2.1 Google Cloud Messaging

Google announced Cloud-to-Device-Messaging (C2DM) system in Google I/O 2010 as a push mechanism for Android applications. It gradually became more efficient and was renamed to Google Cloud Messaging (GCM) in Google I/O 2012[2]. The new version of GCM has received a lot of improvement such as being cross-platform (support iOS and Chrome) as well as having simplified APIs, and was re-branded to Firebase Cloud Messaging (FCM) in Google I/O 2016. Based on reports presented in Google I/O 2016, Google receives around 2 millions queries per second, and more than 1 million apps have been registered by GCM.

[2] https://developers.google.com/android/c2dm/

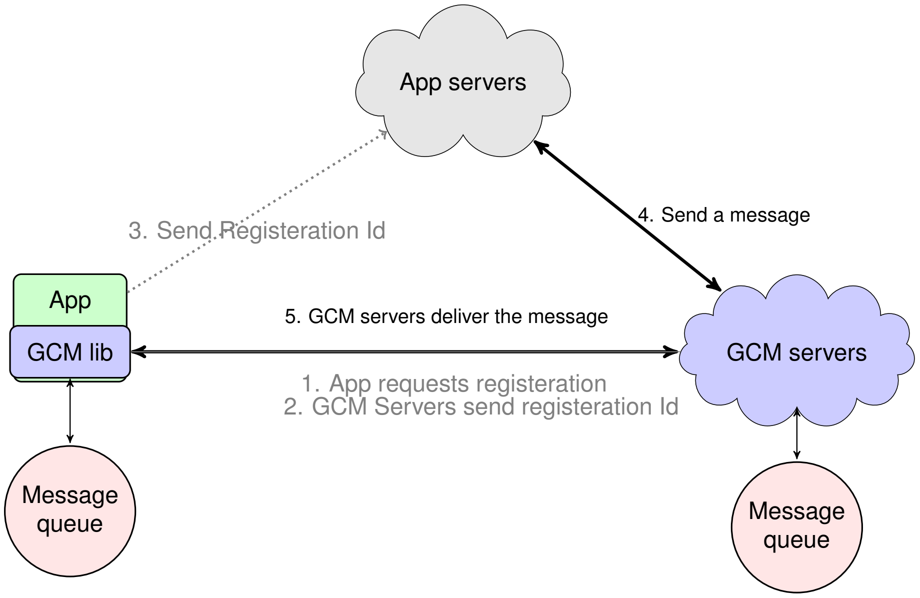

App servers
4. Send a message
3. Send Registeration Id
App
GCM lib
5. GCM servers deliver the message
GCM servers
1. App requests registeration
2. GCM Servers send registeration Id
Message queue
Message queue

Figure 4.1: Google Cloud Messaging Mechanism.

Figure 4.1 shows how the whole mechanism works. First, the application needs to register itself to the GCM servers. After it receives a registration ID from the GCM servers, it sends the registration ID to its server for further communications. Whenever the App server needs to notify its clients, it can send data up to 4KB to a specific registration ID or a group of IDs through the GCM servers. When no Internet access is available on the client device, the messages accumulate in a queue on the server, and synchronize with the client device when it returns online. The connection protocol between App servers and GCM servers can be either HTTP or XMPP[3]. GCM provides a set of APIs for sending messages from servers to applications efficiently and reliably. These APIs can be categorized into 4 classes: *Registration*, an application needs a registration ID to communicate; *Send*, server can send messages to a particular device (registration ID); *Multi-cast*, it is possible to send messages to thousands recipients with a single request; *Time to live*, setting TTL on each request allows GCM to know when to expire a message.

[3] It is a persistent, asynchronous, and bidirectional connection.

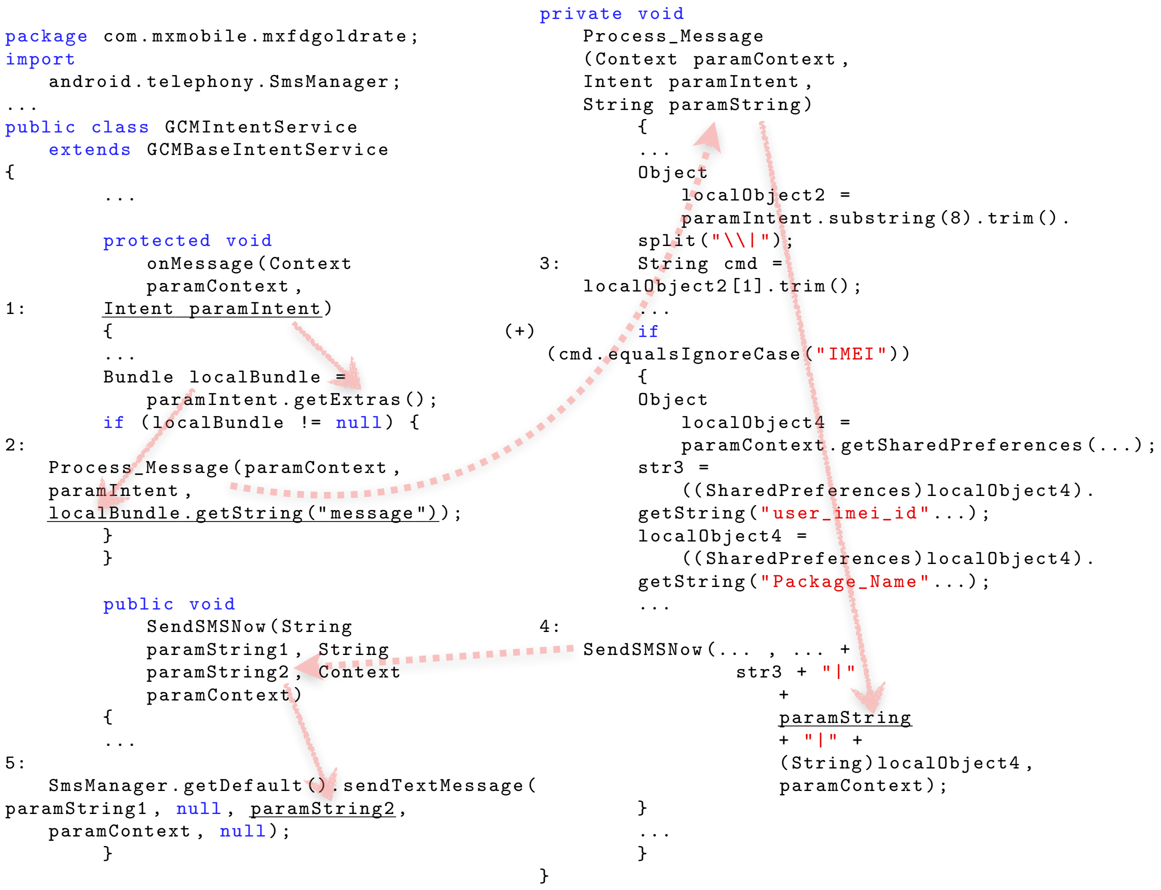
4.2.2 GCM Badware

We describe the two possible cases of use of GCM in badware, namely, bot clients and adware, to better motivate how modeling GCM services can be beneficial in a detection system.

Botnet. Many Android bot clients use unencrypted channels like HTTP to accept messages from command and control bot masters [46]. However, the bot masters can also take advantage of secure popular public services for attacks. Three types of secure services that are frequently exploited by Android bot clients are email over SSL, GCM, and social networks (e.g., Twitter) [251]. By using these services, attackers can launch C&C attacks in a secure way, which is not easily detectable by normal TCP and HTTP traffic analysis. Furthermore, defenders cannot employ simple server blacklisting to mitigate such threats, because the email or GCM servers are used for badware as well as benign applications.

To better explain the misuse of GCM for C&C purposes, Figure 4.2 presents a part of a decompiled `Maxit` backdoor sample [4]. It shows that after the bot client receives a GCM message, the content of the message is accessible through the `Intent` parameter of `onMessage` method (step 1). Then, the data is retrieved from the `Intent` by `getExtras` method and passed to `Process_Message` method which separates the command (step 2, 3) and executes consequent actions based on the command. After doing the action, the bot client sends an SMS to the attacker through `sendTextMessage` (step 5), which is located in `SendSMSNow` method (step 4). The SMS contains the received original GCM message, a retrieved data from `SharedPreferences` (e.g., IMEI) and the package name, which are split by “`|`". Since the attacker might receive many SMS messages, one probable answer to why the bot client sends the original GCM message along with the other information is to make the content of SMS easier for attacker to comprehend. It is obvious that as the received GCM message goes directly to `sendTextMessage`, there is an explicit data flow between the `Intent` parameter of `onMessage` in `GCMIntentService` and the

[4] Badware’s MD5: 157febb16d16e8bcb5ba6564a2f7d320



```
package com.mxmobile.mxfdgoldrate;
import
    android.telephony.SmsManager;
...
public class GCMIntentService
    extends GCMBaseIntentService
{
        ...

        protected void
            onMessage(Context
            paramContext,
1:      Intent paramIntent)
        {
        ...
        Bundle localBundle =
            paramIntent.getExtras();
        if (localBundle != null) {
2:
    Process_Message(paramContext,
    paramIntent,
    localBundle.getString("message"));
        }
        }

        public void
            SendSMSNow(String
            paramString1, String
            paramString2, Context
            paramContext)
        {
        ...
5:
    SmsManager.getDefault().sendTextMessage(
paramString1, null, paramString2,
    paramContext, null);
        }
```

```
private void
    Process_Message
    (Context paramContext,
    Intent paramIntent,
    String paramString)
        {
        ...
        Object
            localObject2 =
            paramIntent.substring(8).trim().
        split("\\|");
3:      String cmd =
    localObject2[1].trim();
        ...
(+)     if
  (cmd.equalsIgnoreCase("IMEI"))
        {
        Object
            localObject4 =
            paramContext.getSharedPreferences(...);
        str3 =
            ((SharedPreferences)localObject4).
        getString("user_imei_id"...);
        localObject4 =
            ((SharedPreferences)localObject4).
        getString("Package_Name"...);
        ...
4:  SendSMSNow(... , ... +
                str3 + "|"
                    +
                    paramString
                    + "|" +
                    (String)localObject4,
                    paramContext);
        }
        ...
        }
}
```

Figure 4.2: A part of Backdoor.AndroidOS.Maxit.a badware, which uses GCM for C&C.

sendTextMessage API. However, if the action is decided based on an if condition, the flow would be implicit. For example, in (step +), if the command equals to IMEI, the badware retrieves the identification number of the device from SharedPreferences so that there is an implicit flow between onMessage and getSharedPreferences.

Adware. Opposite of the aforementioned deliberate misuse of GCM, it might happen that GCM is exploited by badware indirectly. For instance, many adlibraries such as Revmob, Airpush, Leadbolt, Domob and Cauly [2] use GCM to notify users whenever a new advertisement has to be shown. As the purpose of adware is showing advertisement to receive benefits, these adlibraries might be embedded in adware to display unwanted ads when the user is online. As a result, GCM is unintentionally exploited as part of such

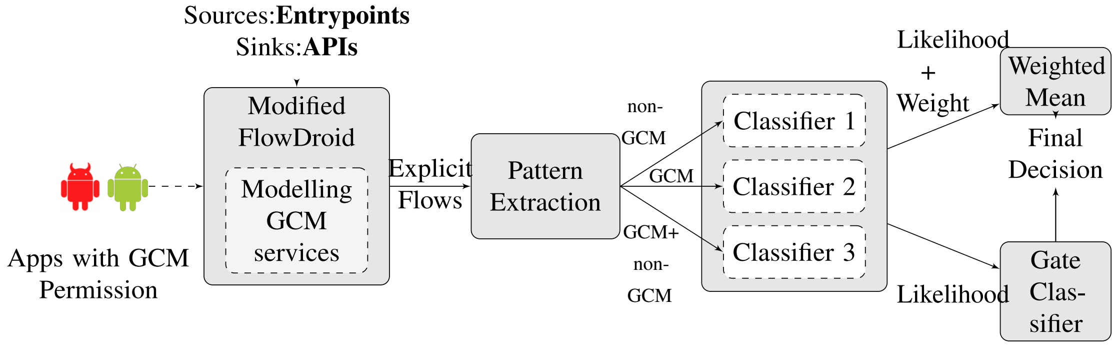


Figure 4.3: Overview of our approach.

malicious activities.

4.3 System Design

The architecture of the proposed system is depicted in Figure 4.3. First, we look into Android applications requesting GCM permission. Second, Flowdroid is used to extract the flows that are originated from GCM. As Flowdroid did not natively analyze GCM flows, we adapted it in order to support GCM callbacks (§4.3.1). Third, the output of Flowdroid is used to extract a number of features that describe the explicit flows. These features are subdivided into two sets, namely GCM and non-GCM categories based on the corresponding services (§4.3.2). Accordingly, a classification function can be learnt by associating each flow to the type of applications it has been extracted from, i.e., badware or goodware. Classification is performed in different ways to verify the effectiveness of GCM features (§4.3.3). In the classification step, we build different models where each model provides a likelihood (between 0 and 1) denoting the degree of maliciousness of an app, which is subsequently thresholded to make the final decision on whether the application is badware or not.

4.3.1 Modeling GCM service

GCM base classes were not supported in FlowDroid [49] because GCM is a part of Google Play Services, and not of the Android Framework. Hence, in order to handle data flows in GCM classes, lifecycles of GCM classes have to be modeled in FlowDroid. Two common classes that have been employed in many applications in the past, are GCMBaseIntentService and GCMListenerService. The FirebaseMessagingService class has been introduced recently so it takes some time to be integrated in some applications. The GCMBaseIntentService class has been deprecated since September 2014, but there are still a lot of applications that have implemented this class. The GCM classes and methods are listed in Table 4.1. The methods are used for different purposes like registration, error handling, and message reception. The application needs to declare a GCMReceiver, which is a kind of BroadcastReceiver so that it delivers messages to GCM base classes. Two important methods that are called during receiving messages are onMessage in the GCMBaseIntentService class, and onMessageReceived in the GCMListenerService. Data flows from parameters of these services can represent command-action behaviors.

To model lifecycles, FlowDroid builds a custom entry point. This entry point is essentially a main() method that emulates the behavior of the Android operating system and framework. As a consequence, the data flow tracker itself can process the app as a standard Java program with a main() method, albeit it still uses the Android framework through calls to library methods. In the basic version of FlowDroid, this dummy main method contains calls the lifecycle methods of activities, services, content providers, and broadcast receivers. Our extension adds calls to the specific methods of the GCM service classes.

One could argue that the GCM base classes such as GCM Receiver are implemented as normal classes inherited from BroadcastReceiver. Therefore, correctly modeling broadcast receivers would be sufficient, because the implementation of GCMReceiver already fully specifies how and when methods such as onMessageReceived are called.

Table 4.1: GCM services lifecycle.

Base Class	Methods
GCMBaseIntentService	void onDeletedMessages(android.content.Context,int) void onError(android.content.Context,java.lang.String) void onMessage(android.content.Context,android.content.Intent) void onRecoverableError(android.content.Context,java.lang.String) void onRegistered(android.content.Context,java.lang.String) void onUnregistered(android.content.Context,java.lang.String)
GCMListenerService & FirebaseMessagingService	void onDeletedMessages() void onMessageReceived(java.lang.String,android.os.Bundle) void onMessageSent(java.lang.String) void onSendError(java.lang.String,java.lang.String)

With this approach, the GCM framework would be treated as part of the app and would be analyzed together with it. For performance reasons, we, however, chose a different approach. We treat the GCM framework classes as black boxes and instead add explicit models for their interfaces. In other words, we consider the GCM framework as a part of the Android operating system and abstract away from it, effectively reducing the size and complexity of the code to be analyzed.

4.3.2 Feature Extraction

Since `FlowDroid` supports detecting the desired flows, as a next step, proper sources and sinks should be provided for, and then run on various applications to extract existing flows. There are two possible ways that source of flows can be defined, i.e., parameters of callbacks, and APIs that retrieve information. As far as we aim to understand what actions are performed when the GCM callbacks are invoked, we consider callbacks as sources in our evaluation to be able to show the power of flows originated from GCM callbacks compared to the rest of Android callbacks. It is worth to mention that, although considering source APIs can provide more information about the semantics of applications, it makes the feature extraction step much slower so we avoided to use them in the proposed system. For sinks, we consider all sink APIs proposed by SUSI [215], which were extracted from Android 4.2 and contain 8,287 APIs.

After the flows are extracted by FlowDroid, they are mapped to a feature vector in which features are flows and their values are the total number of each flow in an application. In other words, we count how many flows there are between a specific pair of source and sink. In flows, the sources are the parameters of callbacks and the sinks are the name of APIs as well as their corresponding package. For example, in the sample in Figure 4.2, there is an explicit flow (by following the red lines) in which the source is `android.content.Intent` in `onMessage` method, and the sink is `sendTextMessage` in `SmsManager` package. Therefore, the feature is in the following format:

$$\texttt{onMessage(android.content.Intent)} \rightsquigarrow \texttt{SmsManager.sendTextMessage}$$

We name such features as “Complete Source/Sink" in the evaluation section(§4.4.2). However, It is common that obfuscation techniques [179, 217] affect the name of some callbacks. Simple method renaming is applied by ProGuard, a popular tool shipped with the Android SDK. For example, we observe the fact in our experiments that `onMessage` callback has different names such as “`a`" or “`nybkaxzg`" in some applications, but the parameters type like `android.content.Intent` are intact. In addition, the sink might be package specific like `startActivity` API from `com.qihoo.psdk.app.QStatActivity` which is the name of an activity in an App. Therefore, we represented the flows to a short format in which we just consider the parameters of callbacks from the sources and API names from the sinks. Note that API method names are usually not affected by obfuscation techniques. So, for the same above example, the feature is in the following format:

$$\texttt{android.content.Intent} \quad \rightsquigarrow \quad \texttt{sendTextMessage}$$

We call this type of features as “Abstract Source/Sink". Although the latter consideration looks loosing some information (e.g., all of `onError`, `onRegistered` or `onUnregistered` methods have `java.lang.String` parameter), we show that they can achieve better result using a smaller number of features compare to “Complete Source/Sink" representation. The underlying reason is that using a more compact (and less noisy) feature representation typically mitigates the so-called problem of *overfitting*, facilitating the task of learning an accurate classification function [61].

To evaluate the effectiveness of GCM features, we divide the features into two sets, i.e., GCM and non-GCM. If a flow is originated from GCM callbacks, we consider it as a GCM feature, otherwise, as a non-GCM feature. So the feature vector is made as follows:

$$fv_i = \begin{Bmatrix} \#f_i & if\left(f_i \in ef\right) \\ 0 & Otherwise \end{Bmatrix} \tag{4.1}$$

Where f refers to a feature, i is the feature index, ef means the extracted features from a sample, and # refers to the frequency of a feature. The equation 4.1 shows if a sample contains a flow, we compute the number of times the flow exists and consider it as the feature(flow) value, otherwise, we consider 0. The following matrix shows an example of the final set of feature vectors, where each row is a feature vector for a goodware/badware, and each column is the frequency of a feature. To separate the features, we prefix them with “g" and “ng", which respectively refer to GCM and non-GCM flows.

$$\left\{ \begin{array}{ccccc} & g\_src_1\_snk_1 & ... & ng\_src_2\_snk_1 & ng\_src_3\_snk_4 \\ B_1 & 3 & ... & 1 & 0 \\ B_2 & 2 & ... & 4 & 1 \\ ... & ... & ... & ... & ... \\ G_1 & 1 & ... & 0 & 0 \\ ... & ... & ... & ... & ... \end{array} \right\}$$

4.3.3 Classification

Regarding the nature of the task at hand, binary classification algorithms are powerful options to help us to discriminate badware from benign applications. Over the past years, a large number of classification techniques have been proposed by the scientific community, and the choice of the most appropriate classifier for a given task is often guided by previous experience in different domains, as well as by trial-and-error procedures. However, some classifiers like SVM and ensemble decision trees (e.g., Random Forest and Extra Trees [130]) have shown high performances in a variety of tasks [125].

To simplify the learning task and reduce the risk of overfitting, we exploit feature selection to reduce the feature set size by removing irrelevant and noisy features from our sets. In particular, as done in [39], we compute the so-called *mean decrease impurity* score for each feature, and retain those features which have been assigned the highest scores.[5]

We combine the obtained decisions of single models (see Fig. 4.3) using a multiple classifier system (MCS) [163,204]. The underlying reason is that MCSs do not only often improve classification accuracy with respect to the combined classifiers, but also provide some degree of robustness against evasion attempts [60, 131]. One of the simplest and widely-used MCS fusion rule is the weighted average:

$$L_c = \frac{\sum_{i=1}^{n} \left(W_i \times L_{c_i} \right)}{\sum_{i=1}^{n} W_i}, \tag{4.2}$$

where n is the number of single classifiers, L is the likelihood of each single classifier, W is a weight assigned to a single classifier, and c refers to each class label. In this approach, a specific weight is assigned to each single classifier output, usually based on the performance of the classifier, and each weight is multiplied by the predicted class likelihood obtained by the classifier. Finally, the class labels are assigned based on the average of the achieved weighted likelihood. Another common MCS technique is passing the likelihood of single models to a gate classifier to make the final decision, which we call it two-tier classification technique. The gate classifier is thus trained in the same way as the flow classifiers, but its input is a feature vector whose components are the output likelihoods of the individual classifiers.

Overall, we build one classifier trained on the GCM features, one classifier trained on the non-GCM features, and a third classifier where all the features, GCM and non-GCM, are used. We observed a degree of complementarity among classifiers, as just a portion of the misclassified samples by one of the classifiers, is misclassified by the other classifiers, the rest of them being correctly classified by the other classifiers. Hence, this motivates the fusion of classification decisions by MCS techniques to combine the

[5] Note that this technique is often referred to also as Gini impurity or information gain criterion.

prediction at the score level. This fusion makes the final decision unbiased between the individual classifiers, which helps improving the final decision. Therefore, we combine the predictions of single classifiers with the weighted mean and the two-tier techniques.

4.4 Experimental Analysis

In this section, we address the following research questions:

- How much discriminatory power do flows from GCM callback sources add to a badware classifier in contrast to only using non-GCM sources (§4.4.2)?
- Is the approach able to predict never-before-seen badware (§4.4.2)?

Before addressing these questions, we discuss the data and the experimental settings used in our evaluation (§4.4.1).

4.4.1 Experimental Setup

To evaluate our approach, we have collected more than 15,000 goodware and 15,000 badware apps from McAfee and VirusTotal[6] sources. The McAfee dataset has been released to the authors on the basis of a research agreement during the period from 2014 to 2016. However, all of the gathered samples are first seen by VirusTotal between 2011 and early 2016. Since this work focuses on analyzing the effects of modeling GCM data flows on badware detection, we filter out all apps that do not use GCM. We consider an app to use GCM if it uses the `com.google.android.c2dm.permission.RECEIVE` permission. We found that slightly less than 10% of our initial set of apps use GCM and were retained. However, checking whether the GCM permission is present is not sufficient, because the app might be overprivileged [121]. Therefore, as a complementary check, we discarded all those of applications that did not have at least one flow from a GCM-related source, i.e., a parameter of a GCM callback method. To obtain the flows,

[6] http://www.virustotal.com

we ran FlowDroid for up to 10 minutes per app on a server with 64 Intel Xeon E5-4560 processor cores running at 2.7 GHz and 1 TB of memory. Note that we limited the maximum heap size allotted to FlowDroid to 250 GB. If the analysis did not complete within this time budget, the app was discarded as well. With these constraints, 1,058 benign and 1,044 badware apps remained for further analysis. Based on the naming convention[7] by VirusTotal, half of the badware are adware, and the rest are trojan.

We evaluate our approach on this set of samples through a 10-fold cross validation, to provide statistically-sound results. In this validation technique, samples are divided into 10 groups, called folds, with almost equal sizes. The prediction model is built using 9 folds, and then it is tested on the final remaining fold. The procedure is repeated 10 times on different folds to be sure that each data point is evaluated exactly once. For the data analysis, we used a laptop with a 2 GHz quad-core processor and 8GB of memory. The whole data analysis code was written in Python, and the main employed helper library is scikit-learn.[8]

Two metrics that are used for evaluating the performance of our approach are the False Positive Rate (FPR) and the True Positive Rate (TPR). FPR is the percentage of goodware samples misclassified as badware, while TPR is the fraction of correctly-detected badware samples. A Receiver-Operating-Characteristic (ROC) curve reports TPR against FPR for all possible decision thresholds.

4.4.2 Results

To better understand the effectiveness of our approach, we evaluate it on the set of Android applications described in Section 4.4.1. To recap the overall approach, we need three single classifiers to make three models on GCM flows, non-GCM flows and the combination of GCM and non-GCM flows. As a first step to better motivate the selection of single classifiers, we use all the three well-performed classifiers, namely SVM, Random Forest and Extra Trees (§4.3.3) to make five models (three single models

[7] https://github.com/ManSoSec/Auto-Malware-Labeling

[8] http://www.scikit-learn.org

plus two MCS models) to see which one provides a better performance. As a result, Extra Trees achieved the higher area under ROC curve so that we select it as the main classifier for the first step of classification (see Figure 4.4). As a consequent step, two MCS techniques, namely weighted mean (MCS-WM) and two-tier (MCS-TT) (§4.3.3) are applied to improve the performance. For the MCS-WM, based on the performance of single classifiers, we assign weights of one, two and three to GCM, non-GCM and the combination of GCM & non-GCM models respectively. For MCS-TT, the output of single classifiers are passed to a gate classifier, which is SVM in our approach.

Our results are summarized in Table 4.2. To better discuss what we explained in Section 4.3.2 about the feature representation, we provide two sets of evaluations on “abstract source/sink" and “complete source/sink". In the “Measures" column, “# Features" shows the number of features used for classification while the numbers in parenthesis refer to the number of selected features. The value of each “FPR" is reported both in terms of the percentage, and in terms of the total number of misclassified goodware (in parenthesis). As is shown in the table, considering GCM-based flows alone is not a proper replacement for a traditional data flow analysis based on non-GCM flows. This is simply because GCM flows represent a small portion of the application behavior. Nonetheless, reported results clearly show that adding GCM flows to the normal flow set containing the non-GCM data flows can be helpful in detecting badware using GCM as part of the malicious behavior. In fact, when we combine GCM features with non-GCM features, they improve the performance, compared to when GCM features are ignored. In the case in which the features (GCM and non-GCM) are stacked, FPR decreases 1% and TPR is improved about 2%. Moreover, the improvement is more observable in the case of MCS-WM in which FPR and TPR respectively recover about 1.9% and 2.4%. In the case of MCS-TT, FPR decreases more, namely 2.2% while TPR has a small improvement of 1.2%. While these numbers seem small, static analyses on Android apps are usually performed on a very large scale, e.g., on complete app stores. If you consider the Google Play Store which contains over 2 million applications, improving the detection rate by 2.4% means that more than 24,000 new, previously undetected pieces of badware are discovered. Lowering the FPR by 1.9% means that 19,000 applications less are flagged as potentially malicious and, consequently, no longer need to be reviewed by human security specialists. On this scale, our proposed improvements greatly improve the state-of-the-art in Android app scanning.

Although there are still some misclassified samples (§4.4.2), we could successfully detect some bot samples based on GCM channel that were not tagged as badware by just relying on non-GCM flows. Last but not least, the results by the proposed “abstract source/sink" features are preferable over “complete source/sink", because they need to consider a significantly lower number of features and can thus be computed more efficiently.

Table 4.2: Classification results of extra tree after feature selection on a set of 1058 benign and 1044 malicious apps.

Measures	Single Classifier on Flows			Multiple Classifiers	
	GCM	non-GCM	non-GCM + GCM	Weighted Mean	Two-Tier
Abstract Source/Sink : Parameter $\rightsquigarrow$ API					
# Features	498 (74)	7,539 (1,162)	8,037 (1,215)	-	6
TPR	89.37%	90.71%	92.72%	93.10%	91.95%
FPR	9.735%(103)	6.049%(64)	5.009%(53)	4.159%(44)	3.781%(40)
Complete Source/Sink : Method.Parameter $\rightsquigarrow$ Package.API					
# Features	3,322 (452)	36,219 (3,892)	39,541 (4,561)	-	6
TPR	89.94%	91.19%	92.53%	93.10%	92.53%
FPR	11.437%(121)	6.805%(72)	5.482%(58)	4.537%(48)	3.686%(39)

Misclassified Samples

We focus here on the proposed MCS architecture, which achieved the best results, and investigate some of the reasons behind its classification errors. As a first step, we checked again the groundtruth labels of all samples by VirusTotal three month after we gathered the last set of samples in our dataset and assigned a new groundtruth. In this way, we built the model with the original groundtruth and then checked the class of misclassified samples with the new groundtruth. Interestingly, we noticed that 4 out of 72 misclassified badware were not labeled as malicious based on the new groundtruth. Moreover, 14 out of 44 misclassified goodware are labeled as badware based on the new groundtruth where all of the 14 samples are labeled as adware. So based on the new groundtruth, we classified 18 (4+14) unknown samples correctly. Among the rest of misclassified badware, 32 of them are adware, and the rest are trojan. The misclassified benign samples need further analysis as there might be some other 0-day badware among them because there are many samples in our analysis from 2016. Another source of misclassification can be the use of obfuscation techniques like dynamic code loading, multi-level reflection, JavaScript and packing. We did not address those techniques in this work as the main focus of this work is modeling GCM services as complementary features.

Furthermore, we explored the features that might contribute the most in the misclassification by computing the median of the feature values in both the sets of correctly-classified and misclassified samples. Some of the features with the highest difference in the median between the two sets are summarized in Table 4.3. The table shows how the classifier might be misled by reducing or adding a specific flow. To point out some of the flows, the ones from GCM methods to `notify` and `Log.v` APIs have higher values in the undetected badware and lower values in the misclassified goodware. It is worth mentioning that the total number of flows alone cannot be representative of the class of applications because both the goodware and badware with almost the same number of flows are present in our dataset (see Figure 4.5). However, there are some goodware that contain higher number of flows and the fact is observable in the figure in the range of 10^3 and 10^4.

4.4.3 Discriminative Patterns

To be more informative, it is worth describing some of the important GCM features that contributed the most in the classification. Figure 4.6 shows the top 20 sink APIs that are performed after a GCM message received in the device and the message in a way passes to those APIs. Each of those APIs can reveal some useful information about an action that might reveal a sign about a malicious activity. As an example, 25 badware samples execute the `sendTextMessage` method based on the content of received GCM messages while no goodware shows this dependency between a received GCM message

Table 4.3: Features that contribute the most in misclassification. Minus/plus refers to reduction/addition of a feature.

+/-	Feature
From correctly-classified goodware to false positives	
+	non-GCM : android.content.Intent ⇝ putExtra
+	non-GCM : android.os.Bundle ⇝ putString
+	non-GCM : android.os.Bundle ⇝ onCreate
-	GCM : android.content.Context ⇝ notify
-	GCM : android.content.Context ⇝ v
From true positives to undetected badware	
+	GCM : android.content.Context ⇝ notify
+	GCM : android.content.Context ⇝ v
-	non-GCM : android.os.Bundle ⇝ onCreate
-	non-GCM : android.content.Intent ⇝ putExtra
-	non-GCM : android.view.KeyEvent ⇝ onKeyDown

and an outgoing text message. Other kinds of suspicious actions are `openConnection` and `setRequestMethod` that exist in higher number of badware samples compared to goodware. These patterns can model downloading payload or performing DDoS attacks as the flow can show a command-and-control structure. The attacker sends a URL from the control server to his bots using GCM and the bots then perform requests against the received URL.

In adware, it is hard to say how each single pattern alone can divulge a malicious activity as the same adlibrary, embedded in adware, can be used in goodware as well. However, the experiments showed that combination of GCM features with others can contribute in the detection of adware, as this links the presence of the adlibrary to other facts about the application to provide more information about the context in which the library is used.

4.5 Limitations

Considering that our approach is built on top of FlowDroid, our system inherits its corresponding limitations. First, it has difficulty to track API calls that are employed by reflection techniques. Second, it cannot follow flows to the native code as it is a flow analysis system for Java. Third, dynamic code loading techniques should be another issue as FlowDroid is a static analysis technique and an attacker can download a code from internet as well as load a code from a local storage, and then load it during runtime. Moreover, FlowDroid doesn't handle inter component communications. While some recent papers have partially addressed the aforementioned issues [167, 168, 213, 216], there is a need to push forward handling these issues to the next stage.

Apart from static analysis limitations, there are possible evasion techniques against machine learning like mimicry attacks. For example, if the detection system didn't consider the semantic, an attacker can simply inject some dead code to evade the detection system [258]. Although we didn't evaluate our approach against these kinds of attacks, an adversary has to modify particular flows in application to evade our system, which is not easy and needs a lot of efforts.

4.6 Comparison with Related works

As far as the main purpose of our approach is application analysis for badware detection, we describe some prominent approaches in the application analysis area with a highlight on those that focus on GCM and adlibraries.

Badware Analysis. There are also a quite good number of approaches that proposed different static and dynamic analysis techniques for badware classification. They are mostly based on machine learning while the difference derives from the feature extraction step. Some of these approaches vet badware detection like Drebin [48], DroidAPIMiner [36], MudFlow [53], AppAudit [267], while the others just concentrate on badware family classification like DroidScribe [88] and Dendroid [243]. Moreover, there are some systems that generalize their approach for both malware detection and classification such as DroidMiner [275] and DroidSIFT [280]. Apart from the aforementioned systems that consider badware in generic cases, there are also some researches that target some specific kinds of badware like the one that provides a solution for detection of logic bombs [127] in Android applications, or another one that extracts potentially suspicious runtime values such as premium SMS numbers or blacklisted URLs [216] to thwart evasion techniques.

Advertisement Libraries Analysis. On the evaluation of advertising libraries, there are some approaches like AdDroid [202] and AdSplit [235] that contributed on isolating advertising libraries from host applications (e.g., to create fault isolation). A common way for adversary to monetize the adlibraries is to repackage free benign applications with injected adlibraries, and then earn ad revenue as explained by Zhoe et. al [288]. They proposed an approach to decouple the core of applications from other modules (e.g., adlibraries) based on program dependency graph to detect repackaged applications. Another work [240] examines the effects on user privacy in thirteen popular Android ad providers by reviewing their use of permissions.

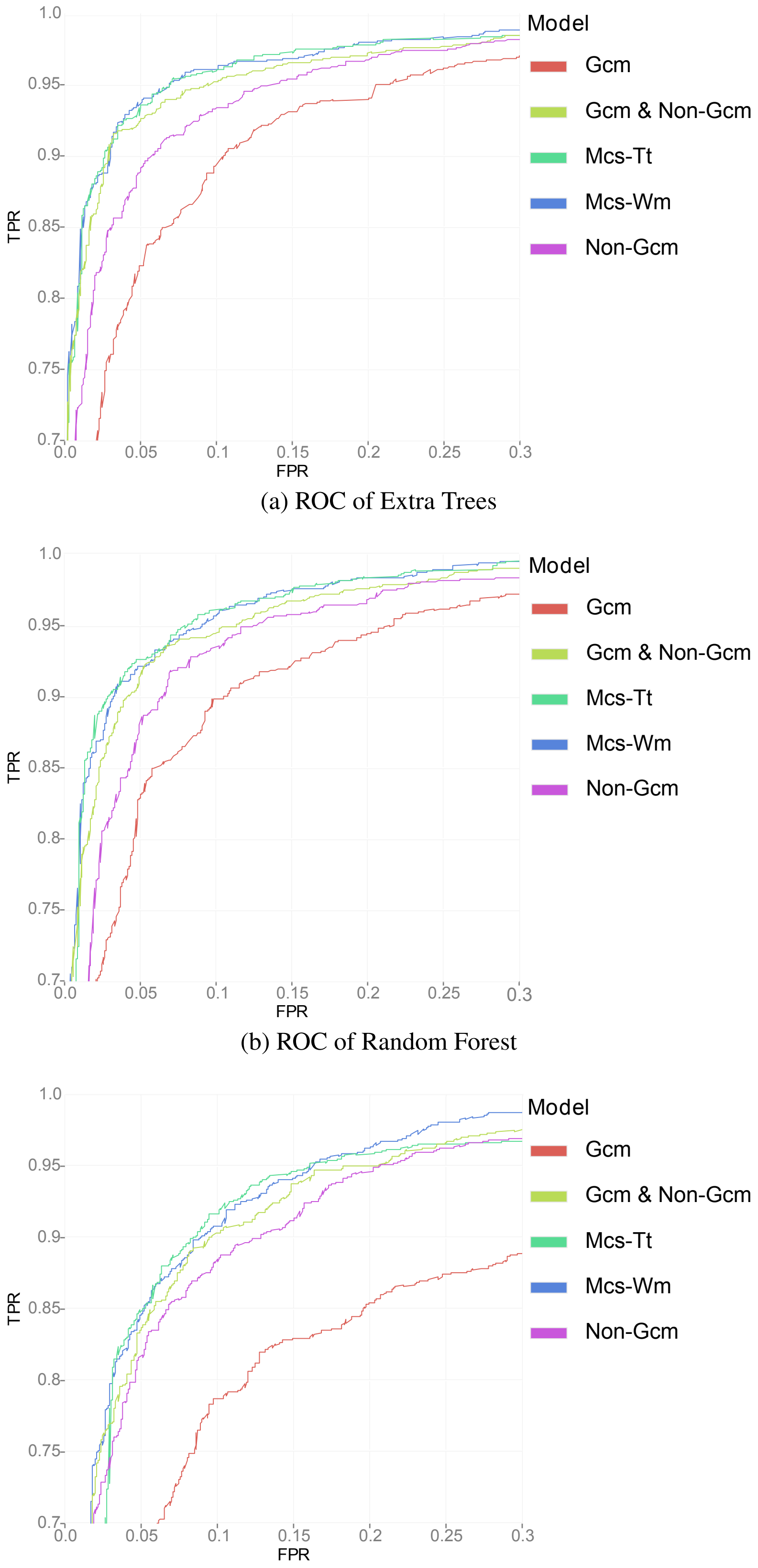
GCM Services Analysis. There are just a few works that analyzed GCM services for security objectives. The most prominent example is the one that reported some vulnerabilities in GCM mechanism [169] by which an adversary can steal sensitive user data of popular applications like Facebook, and command the devices. A following work, called Seminal [77], provides an automation to find vulnerabilities in applications using push notification services. Apart from vulnerability analysis, other researchers [285] showed how attackers might exploit push notification services like GCM to create a cloud-based push-styled mobile botnet. However, they didn't propose any concrete analysis/defense solution except advising either monitoring the network traffic or verifying the combination of GCM permission with others.

As an overall comparison with the previous approaches, opposite of the other works, this paper aimed to model the behaviors of GCM services in Android applications statically to more effectively discriminate badware from benign applications. Moreover, based on the best of our knowledge, we are the first to use the MCS paradigm for Android badware detection, which can help improving the performance of the single classifiers.

4.7 Conclusions and Future Work

In this chapter, we modeled Google Cloud Messaging in Android applications to be able to detect flows from GCM services, which consequently helps analyzers to investigate security issues related to these services automatically. Consideration of the GCM services is important because of the advent of GCM in badware where GCM acts as a C&C channel. To assure how much this consideration can be beneficial, we evaluated the effect of data flows from GCM services for badware detection. Our results indicate that GCM features help to more effectively discriminate badware using the GCM mechanism from benign applications, compared to when they are ignored. The proposed approach benefits from the MCS approach which was proved to be more resilient to evasion in computer security, so we expect the same behavior. As a future plan, it is worth to extend this work to support every kind of push services as they might be exploited more extensively (e.g., Baidu Cloud Push service was abused in a badware[9]).

[9] http://b0n1.blogspot.co.uk/2015/03/remote-administration-trojan-using.html



(a) ROC of Extra Trees

(b) ROC of Random Forest

(c) ROC of SVM

Figure 4.4: ROC curves of different classifiers. The best result was achieved by Extra Tree classifier.

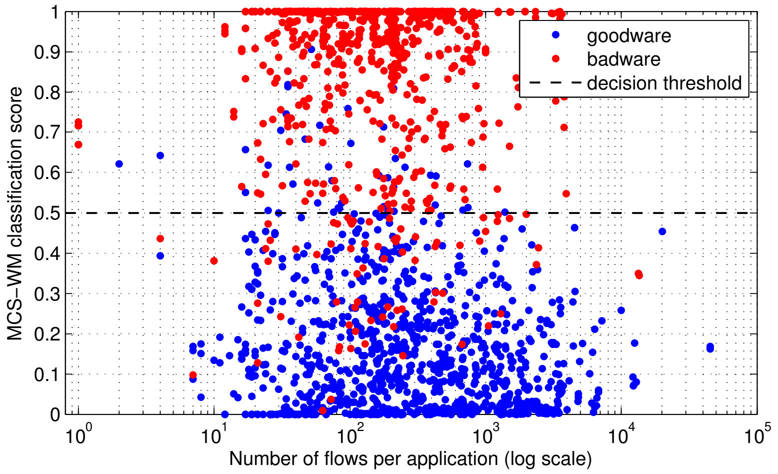


Figure 4.5: The realationship between the total number of flows in applications and the classification score.

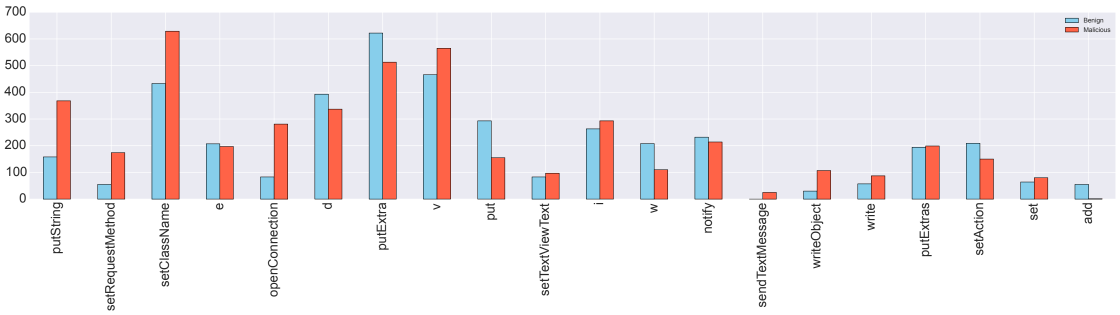


Figure 4.6: The figure shows 20 discriminative actions, which are among top selected features. These actions are sink APIs in the data flows that are originated from GCM services). `e`, `d`, `v`, `i`, `w` are `log` methods.

Chapter 5

Fast and Accurate Classification of Obfuscated Android Malware

5.1 Introduction

For both malware detection and family identification, we strongly prefer light-weight and scalable methods to cope with the numbers of apps, both benign and malicious.

In general, static analysis techniques are computationally cheaper than emulation-based dynamic analysis; unfortunately, many static analysis techniques are easily thwarted by obfuscation, which is becoming increasingly common on Android [217]. Family identification in particular also suffers from the widespread code reuse in malware, which leads to different malware families sharing code and entire modules.

To address these challenges, we introduce DroidSieve, a system for malware classification whose features are derived from a fast and scalable, yet accurate and obfuscation-resilient static analysis of Android apps. DroidSieve relies on several features known to be characteristic of Android malware, including API calls [36, 48, 275], code structure [243], permissions [283], and the set of invoked components [48]. In addition, DroidSieve performs a novel deep inspection of the app to identify discriminating features missed by existing techniques, including native components, obfuscation artifacts, and features that are invariant under obfuscation. In particular, we make the following contributions to the state of the art:

- We introduce a novel set of features for static detection of Android malware that includes the use of embedded assets and native code; it is at the same time robust and computationally inexpensive. We evaluate its robustness on a set of over 100K benign and malicious Android apps. For detection, we achieve up to 99.82% accuracy with zero false positives. The same features allow family identification with an accuracy of 99.26%.

- We analyze the relative importance of our features and demonstrate that artifacts introduced by state-of-the-art obfuscation mechanisms provide high-quality features for reliable detection and family identification. Moreover, we show that there is a small set of features that perform consistently well regardless of whether they are derived from obfuscated or plain malware.

The rest of this chapter is organized as follows: We first motivate our choice of features by briefly reviewing obfuscation techniques in Android malware (§5.2). We then describe our two main classes of features as well as employed learning algorithms (§5.3) before presenting our experimental setup and results (§5.4). Finally, we review related work (§5.5) and conclude (§5.6).

5.2 Obfuscation in Android

We now briefly review the state of the art in Android obfuscation as it motivates our work. Thorough taxonomies of software obfuscations are available in the literature [80, 231].

String Obfuscation. Recent approaches to fingerprinting malware have made use of string-based features such as permissions and apps/package names [36, 48, 171]. Some strings, such as the declaration of application permissions, follow a strict syntax and must appear in the clear; other strings, such as names and identifiers, can be easily randomized or encrypted [74, 179].

Native Code. Native code is also frequently used to offload malicious functionality from the main Dalvik executable (DEX) to dynamically linked libraries or other executables (ELF files), which are then invoked at runtime.

Dynamic Code Loading. Native code and additional Dalvik bytecode can be loaded from a library included in the app's assets, from another app (collusion attack) or from a remote system after being retrieved at runtime. In our experiments, we found many examples of dynamic code loading, including cases where code was loaded from outside of the app. However, the mere presence of dynamic code loading is not malicious in itself, since many regular software frameworks employ this technique, which makes it even more attractive to malware writers.

Code Hiding. Malware authors often proactively hide malicious components to make the overall application look benign to cursory inspection [45]. For instance, the *GingerMaster* malware hides Bash scripts for its packaged root exploit under innocuous file names such as `install.png` and `gbfm.png` in its resources [241]. Other malicious apps go a step further and use a form of steganography, e.g., by hiding malicious code inside a valid image file [242]. The app loads the image through a seemingly benign action but uses a decoding algorithm to extract a malicious executable payload[1].

Finally, Android malware can also hide its malicious payload in an APK file hosted as a resource of the main app. When the app is executed, the user is lured into installing the hidden APK and the system then dynamically loads the hidden component. In the rest of this chapter, we refer to these apps as *incognito apps*. In a related scenario, the *update attack*, the app just contains a component that downloads and executes a malicious payload from an external server. Such attacks are hard to detect and mitigate as the app misleads the user to grant the additional permissions while pretending to update itself [208].

The aforementioned methods for code hiding can easily be combined with encryption

[1] A recent example is *Android/TrojanDropper.Agent.EP* (*MD5:1f41ba0781d51751971ee705dfa307d2*), November 2015.
`b0n1.blogspot.co.uk/2015/11/android-malware-drops-banker-from-png.html`

to further obfuscate the malicious payloads and decrypt them only at runtime [45]. While encryption makes it harder to assess the component statically, its presence can be detected by measuring the entropy of the component. However, encryption is also commonly employed by benign apps, and during our experiments, we particularly found that many benign apps were using encrypted strings.

Reflection. Reflection is a commonly used feature in various Java frameworks, but it is also a notorious impediment to static analysis, since it may be infeasible to statically determine which code is executed at runtime. As a consequence, malware writers have long discovered reflection for obfuscating sensitive API calls and libraries [45]. In a recent large-scale study, Lindorfer et al. [172] showed that the general use of reflection among apps has increased significantly.

The state of obfuscation on Android has caught up with that on desktop systems, and there are already automatic frameworks available for obfuscating Android app components [74, 179, 217]. Hence, obfuscation now poses a serious challenge for static malware analysis on Android and has to be addressed to achieve robust malware classification.

5.3 Proposed System

The architecture of the proposed system is depicted in Figure 5.1. The detail of feature engineering step is discussed in §5.3.1-§5.3.4, and the classification step is discussed in §5.3.5.

5.3.1 Feature Engineering

We now introduce our proposed set of features for both malware detection and identification of malware families. Based on an analysis of existing malware (§5.3.2), we identify two major classes: *resource-centric* features are derived from resources used by the app (§5.3.3); *syntactic* features are derived from the code and metadata of the

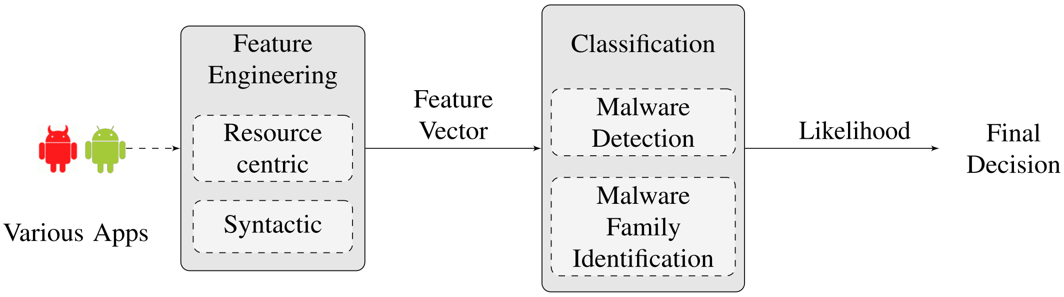


Figure 5.1: Overview of our approach.

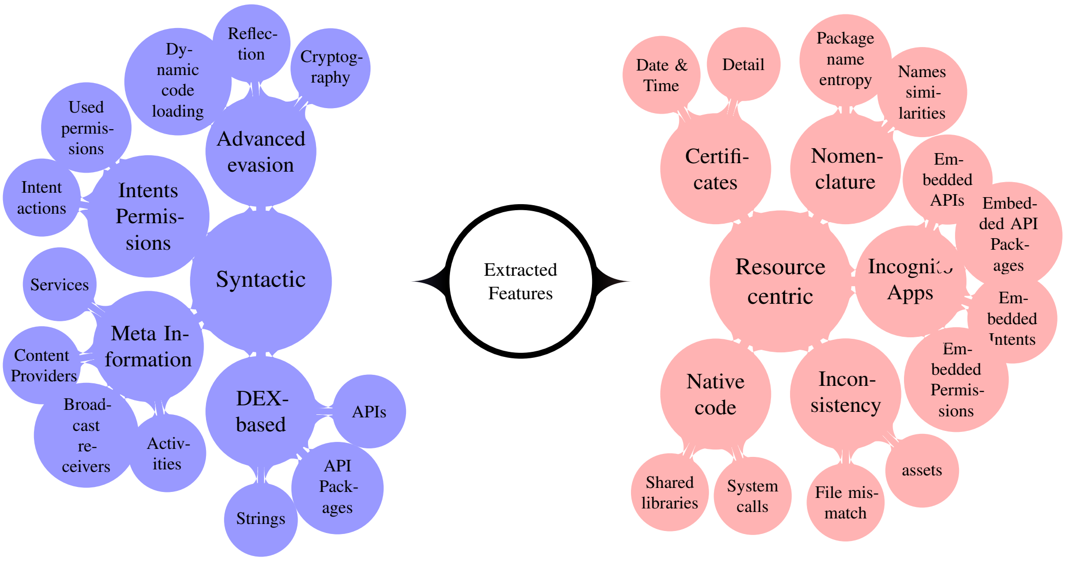


Figure 5.2: Non-exhaustive map of extracted features. The left side shows syntactic features derived from the source code of the app; the right side shows resource-centric features derived from the assets of the app.

app (§5.3.4). A map relating classes of features is shown in Figure 5.2. We use both binary and continuous features. The presence or absence of a particular trait, such as a permission, is encoded as a binary feature; numeric properties, such as string lengths or opcode frequency, are encoded as continuous features.

5.3.2 Prevalence of Features

Robust classification requires a diverse set of features. Features such as API calls are highly relevant for classifying non-obfuscated malware but are susceptible to obfuscation.

The presence of obfuscations may indicate malware, but it is not by itself sufficient to form judgment, since benign software can also use the same techniques for legitimate purposes. Therefore, we propose to employ a portfolio of features that covers both non-obfuscated and obfuscated malware.

As a first step towards selecting effective features, we measured the prevalence of a wide range of features that could be effective at identifying both obfuscated and non-obfuscated Android malware. We hypothesize that features centered around steganography, where the sample hides its malicious payload in its assets, or inconsistent nomenclature of components of an app by a careless malware developer are important features. To test our hypothesis, we run an assessment on a collection of over 100,000 benign and malicious samples from multiple sources. To put our findings in perspective, we also select some features from published works on Android malware identification.

For benign samples, we obtained a dataset of clean apps vetted by McAfee (McGW). For the malicious samples, we relied on two commonly used datasets: the Malgenome Project (MgMW) [293] and the Drebin dataset [48]. We further extended our dataset with the goodware (MvGW) and malware (MvMW) collected by Lindorfer et al. [171]. To measure feature prevalence in obfuscated malware, we also include the recent PRAGuard (PgMW) dataset [179]. The samples in PRAGuard were obtained by obfuscating the samples of the MgMW dataset with techniques such as class and method renaming, reflection, and class encryption, among others.

Table 5.1 summarizes the results of our investigation. We can observe that most of the features are more prevalent in malware than in goodware. In particular, structural and logical inconsistencies are between 5% and 35% more prevalent in malware. In fact, the difference in prevalence of these features is comparable to well-understood features such as permissions, sensitive API calls, and those related to SMS messaging. Thus, inconsistencies are an important class of feature that have not been reported in the literature so far.

Our work also identifies obfuscated malware. In view of this, we also looked for prevalence of features that may hint at obfuscation in the form of reflection or the use of native code. In our study, McGW contained a prevalence higher than either MvGW and most of the malware datasets. This is because McGW is a more recent dataset with samples ranging from 2012 to 2016 and use advanced coding techniques while other datasets with the exception of PgMW are from 2012 and 2013. One may wonder the utility of including these features as a part of the classifier if they cannot be used to classify modern samples. A key assumption that we make here is that the classification model should evolve over time as pointed out recently in the literature [100]. Features that are relevant today will naturally become irrelevant in the future and it is the responsibility of malware analyst to purge obsolete features from the model while retraining. For our experiments, we retain these features as we test our features over a large timespan.

The PgMW dataset deserve special mention as it highlights how standard forms of obfuscation can confound the classification model. For the PgMW dataset, it can be seen that some features that are common in malware can be easily obfuscated. For example, methods that are crucial for the detection of malicious activities, such as communications (*SMS*) or the access to sensitive information (*getSimSerialNum.*), have been nearly eliminated in the obfuscated dataset. Therefore, relying on these features alone when dealing with obfuscation is detrimental to malware analysis and detection. These findings further reinforce our original suggestion of using a diverse portfolio of features for resilient classification.

5.3.3 Resource-centric Features

We propose a set of new features extracted from the app's resources stored in the APK. An excerpt of the resource-based features that we use can be seen in Table 5.1.

The two main guiding criteria that we use for building the set of resource-based features are *structural inconsistencies* and *logical inconsistencies*. Structural inconsistencies refer to the artifacts left behind after hiding a malicious component. Logical inconsistencies refer to the footprints typically left when repackaging a piece of malware as part of a benign app.

Type	Capability	Goodware		Malware				Summary	
		McGW	**MvGW**	**MgMW**	**Drebin**	**PgMW**	**MvMW**	**Goodware**	**Malware**
Logical Inconsistencies	Main Activity	15.01%	8.85%	29.44%	18.71%	29.60%	8.23%	9.31%	13.13%
	Service	43.44%	4.60%	72.62%	54.17%	74.29%	35.34%	7.51%	44.18%
	Receiver	46.23%	13.57%	74.29%	56.06%	75.87%	36.60%	16.02%	45.66%
Structural Inconsistencies	APK File Match	1.77%	0.07%	24.21%	6.51%	24.13%	2.23%	0.20%	5.18%
	APK File Extension Mismatch	1.41%	0.02%	23.89%	6.28%	24.13%	2.22%	0.12%	5.10%
	Image File Extension Mismatch	3.69%	1.48%	19.92%	8.22%	18.17%	1.44%	1.65%	4.82%
Sensitive API	Package: SMS	5.63%	1.92%	20.79%	36.53%	0.00%	57.80%	2.20%	46.82%
	TelephonyManager.getSimSerialNum.	9.24%	4.69%	50.63%	24.06%	0.08%	14.22%	5.03%	16.34%
Permissions	READ_CONTACTS	22.93%	6.26%	36.27%	23.29%	38.8%	17.20%	7.52%	20.71%
	ACCESS_FINE_LOCATION	28.04%	16.40%	34.29%	30.04%	32.30%	15.53%	17.28%	21.38%
Obfuscation	Dynamic Code	32.22%	0.44%	19.60%	6.98%	0.00%	2.04%	2.83%	3.47%
	Reflection	74.08%	39.37%	67.62%	56.04%	99.21%	40.14%	41.97%	49.50%
	Native Code	49.61%	3.69%	54.13%	19.51%	0.16%	6.43%	7.14%	10.15%
	Native Code without ELF	8.10%	0.58%	1.67%	0.70%	0.00%	0.52%	1.14%	0.54%
Total Number of samples		8,041	99,037	1,260	5,560	1,260	10,581	107,078	17,401
Total Number of families		–	–	49	179	49	–	–	–

Table 5.1: Percentages of apps with given properties in the McAfee Goodware (McGW), Malgenome (MgMW), Drebin malware, PRAGuard's obfuscated Malgenome (PgMW), Marvin Goodware (MvGW) and malware (MvMW) dataset. Note that the summary shows the total number of apps after removing overlapping samples.

Certificates. We check whether the times at which the app was signed and at which the certificate was generated are similar. The intuition behind this feature is that automated repackaging tools modify existing apps and sign them using auto-generated ad-hoc certificates before distribution. Thus, if the date when the certificate was created is close to the date on which the app was signed, it can reveal the use of an automated tool for app repackaging. We mark apps where the time difference was below ten minutes. For each certificate, we also build features from the timezone and the common name's string length, which allows to identify similar certificates generates by repackaging tools.

Nomenclature. For each of the components in the app, we verify whether the correct package name is used as a prefix of the components in a package directory which is the usual practice in most apps. If there is a mismatch, we treat it as a potential case of tampering with the original contents of a benign app. Table 5.1 shows an overview of the percentage of samples that exhibit such a mismatch. For each of the package names, we also derive its length and its Shannon entropy, which help to identify automatically generated names.

Inconsistent Representations. We check whether the file extensions match the file contents (as identified by the file header or a magic number) to allow highlighting apps that try to hide shell scripts or ELF binaries as images or other resources. Table 5.1 shows that such inconsistencies are good indicators of malicious intent in some (e.g., Malgenome) but not all (e.g., Marvin) datasets, potentially owing to trends in malware writing and repackaging tools.

Incognito Apps. In some cases the payload of a malicious app is in an APK that is disguised among the assets of the *host* app. To capture this malicious payload, we recursively extract both syntactic and resource centric features for any *incognito* APK and DEX found within the app. We pigeonhole these features under a different category in order to separate these statistics from the ones related to the *host* app. For instance, *permission.INTERNET* counts the static number of accesses to the Internet, while *icg.permission.INTERNET* does the same for the incognito app.

Native Code. We also scan the assets of the app to identify any native ELF files. The files are parsed to extract features from the header and individual sections of the file. We extract the number of entries in the program header, the program header size, and the number and size of the section headers. From individual sections, we extract the flags of the section to understand if they are W (writable), A (allocatable), X (executable), M (mergeable), S (strings), etc. and use them as Boolean features. Within code sections, we also look for instructions invoking critical system calls such as `ioctl`, which is used for Android's inter-procedural and inter-component communication.

5.3.4 Syntactic Features

We present our syntactic features; several of these, such as API calls [36] and permissions [283], are already known to perform well with non-obfuscated malware. We don't claim novelty by including these features. Instead, we use them to build a classifier that is robust against both well-known and modern malware which tends to be increasingly obfuscated. To enrich the set of syntactic features, we propose some new features such as *explicit intents* and additional ones mined from the *meta-information*. These are discussed below. We reiterate here that a combination of diverse features is crucial for robust detection of both plain and obfuscated malware. This is corroborated in Section 5.4.2 where important features come from diverse categories, yet they all rank highly in relation to other features (see Figure 5.3 and 5.4).

DEX-based Features. We tag each method based on the libraries it invokes from the Android Framework (*method tag*). These tags represent the class of APIs used by the method and are encoded as binary features. We also scan the app for the presence string variables in DEX files containing keywords we obtained from reverse engineering malware from the Malgenome data set. For instance, `su` relates to executing code with super user privileges; `emulator` and `sdk` suggest that the app checks for the presence of an emulator.

Intents and Permissions. We parse the Manifest to identify all implicit intents that can be receive from other apps. We also scan the code to identify any explicit intents, which are used to start services within the same app. The count of individual intents is used as a continuous feature for classification. We break down the set of intents into sub-categories for further granularity: (i) intents containing the keywords `android.net.*`, which are related to the connection manager; (ii) intents containing `com.android.vending.*` for billing transactions; (iii) intents that target framework components (`com.android.*`); (iv) all intent actions, beginning with `android.intent.action.*`; and (v), a catch-all category for the reminder intents. Finally, we also extract the set of permissions declared in the manifest of the app.

Meta-information. Apart from the specific type of permission used, we also count the number of Android framework permissions and custom third-party permissions used by the app. The number of times that a permission is used throughout the code is encoded as a feature. Similarly, we count the number of activities, broadcast receivers, content providers, services, and entry points of the app. Entry points are ways in which an app can be invoked or executed.

Evasion Techniques. We further look for techniques that are frequently used to confuse analysis systems, i.e., native code, cryptographic libraries, or reflection. For example, `Ldalvik/system/DexClassLoader` indicates dynamic code loading, `Ljava/lang/reflect/Method` is required for invoking a method through reflection, and any access to `Ljavax/crypto` is a sign for the use of cryptography. For native code invocations, we count the number of times the Dalvik opcode `0x100` is present in the bytecode, which corresponds to loading and executing native code.

5.3.5 Choice of Learning Algorithm

We implemented both malware detection and family identification in DroidSieve using Extra Trees. As alternatives we considered one-vs-all Support Vector Machines (SVM), Random Forests, and eXtreme Gradient Boost (XGBoost). In the past, SVM and Random Forest have been successfully applied to malware detection [48, 237] and they have been shown to have better performance than others after comparing them to 180 classifiers on various datasets [125]. Ensemble tree-based classifiers perform well on many real world settings, however. For example, Extra Tree [130] and Gradient Tree Boosting [147] have been achieving great performance in most of recent “Kaggle” competitions [39] on various domains, including malware classification[2] or spam detection[3].

We use feature selection to restrict the classifier to important discriminating features. A feature is selected when the importance score assigned to the feature by the classifier is higher than the mean of all the features’ scores. For decision trees, this importance is computed from the mean decrease impurity (MDI) where a higher score implies a more important feature.

[2] http://blog.kaggle.com/2015/05/26/microsoft-malware-winners-interview-1st-place-no-to-overfitting/

[3] http://mlwave.com/winning-2-kaggle-in-class-competitions-on-spam/

5.4 Experiments and Results

We implemented our proposed feature set in DroidSieve, a system for static detection and family identification of Android malware. We begin our evaluation by describing our experimental setup and evaluation metrics (§5.4.1). We then address the following questions:

- **Feature Engineering** (§5.4.2): Which types of features are most effective for regular and obfuscated malware?
- **Classification of Standard Samples** (§5.4.3): How effective is DroidSieve in classifying non-obfuscated malware only, and how does it compare to other approaches that address the same problem?
- **Classification of Obfuscated Samples** (§5.4.4): How effective is DroidSieve in classifying obfuscated malware or a mix of non-obfuscated and obfuscated malware?
- **Computational Efficiency** (§5.4.5): Do the computational costs of using DroidSieve allow its application at scale?

5.4.1 Experimental Setup

We mean to evaluate the choice of our features for two distinct problems:

Evaluation Categories. We evaluate DroidSieve along two dimensions, the classification task and the type of dataset. The classification task is either (1) *malware detection* among a set of malicious and benign samples or (2) *family identification* among a set of samples known to be malicious. The type of dataset is either non-obfuscated, obfuscated, or mixed. We use the datasets introduced in §5.3.2 and combinations thereof; details are shown in Table 5.2a.

Evaluation Metrics. For evaluating the classification results, we use the detection rate (DR), the false positive rate (FPR), the accuracy (ACC), and the F_1-score (F1) which is

ID	Dataset Name	Ground Truth	#samples
—	Drebin [48]	Malware	5,560
MgMW	MalGenome [293]	Malware	1,260
PgMW	PRAGuard * [179]	Malware	1,260
McGW	McAfee	Goodware	8,041
McMW	McAfee	Malware	13,289
MvGW	Marvin [171]	Goodware	99,037
MvMW	Marvin [171]	Malware	10,581

(a) Dataset sources

Set	Detection	Family Identification
1	{McAfee Goodware, Drebin}	Drebin
2	{McAfee Goodware, MalGenome}	MalGenome
3	{McAfee Goodware, PRAGuard*}	PRAGuard*
4	{Marvin Goodware, Marvin Malware}	–
5	{McAfee Goodware, McAfee Malware}	–
Hold-out Ratio: 67% Training – 33% Testing		

(b) Dataset combinations

Table 5.2: Overview of chosen datasets for malware detection and family identification. The set of experiments involving obfuscated samples is marked with an asterisk(*). The holdout ratio shows the percentage of samples retained for validation. For the case of Marvin and McAfee malware we retain the splitting given by the authors, otherwise we use a random split.

the harmonic mean of the precision and recall as quality metrics. Detection rate is the correct number of predictions made over the set of malware, whereas accuracy reports the number of correct predictions made after considering both goodware and malware. We only use the detection rate for the case of malware detection and we report this metric together with the false positive rate, i.e., the number of goodware samples wrongly classified as malware divided by the total number of goodware samples in the dataset.

For assessing the performances of the proposed models, we use hold-out validation to avoid overfitting [103]; samples used to fit the model are different from the ones used to validate it. We retained one third of the samples for validation and trained the model on the remaining two thirds of the data. For each sample that was retained, we ensured that we trained on samples from the same category. For malware detection, a category for a sample indicates whether it is benign or malicious. For family identification, a category indicates the name of family. Consequently, we do not have a case of testing on samples from unseen families or categories; this would be an instance of zero-shot learning [198], a problem we consider out of scope for this thesis.

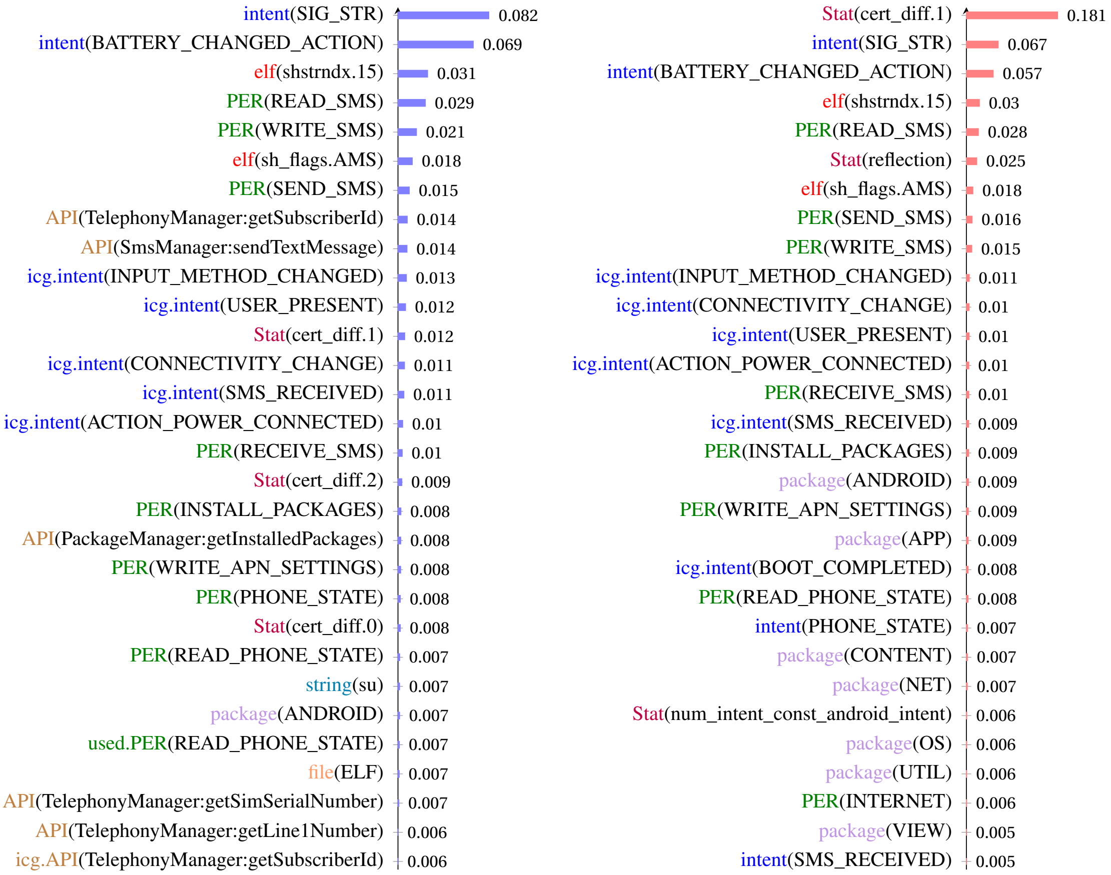
5.4.2 Ranking of Features

We now analyze the quality of our features, ranking them when used on unobfuscated and obfuscated datasets. We expect features that are easily obfuscated to decrease in importance, whereas features that are invariant under obfuscation should remain stable.

We pass the feature vectors for our samples to the *Extra Tree* algorithm and rank them by *mean decrease impurity* [177]. As decision trees split the dataset by considering one feature at a time, it is easy to measure how much *impurity* is introduced in the classification by choosing a particular feature. Note that these rankings are informative and do not dictate our choice of features in all sets of experiments in §5.4.3 and §5.4.4. For classification, DroidSieve uses automatic feature ranking and chooses the top features for the respective training set.

For malware detection, we passed all samples in McGW + MgMW and McGW + PgMW through the Extra Tree classifier. Figure 5.3a and Figure 5.3b depict the top 30 features for these cases, respectively. In the case of McGW + MgMW, these 30 features account for the top 40% features, while in the case of McGW + PgMW these features account for the top 36% features. We repeated a similar experiment for the case of family identification and the top features for samples from MgMW and PgMW are presented in Figure 5.4a and Figure 5.4b, respectively. They denote the top 26% and the top 43% most important features for identifying Android malware families from MgMW and PgMW, respectively.

For both plain and obfuscated malware, it may be seen from Figures 5.3a and 5.3b that permissions (prepended with PER) play an important role in the detection process. Permissions are hard to obfuscate as scrambling them would break the Android



(a) Non-obfuscated malware.

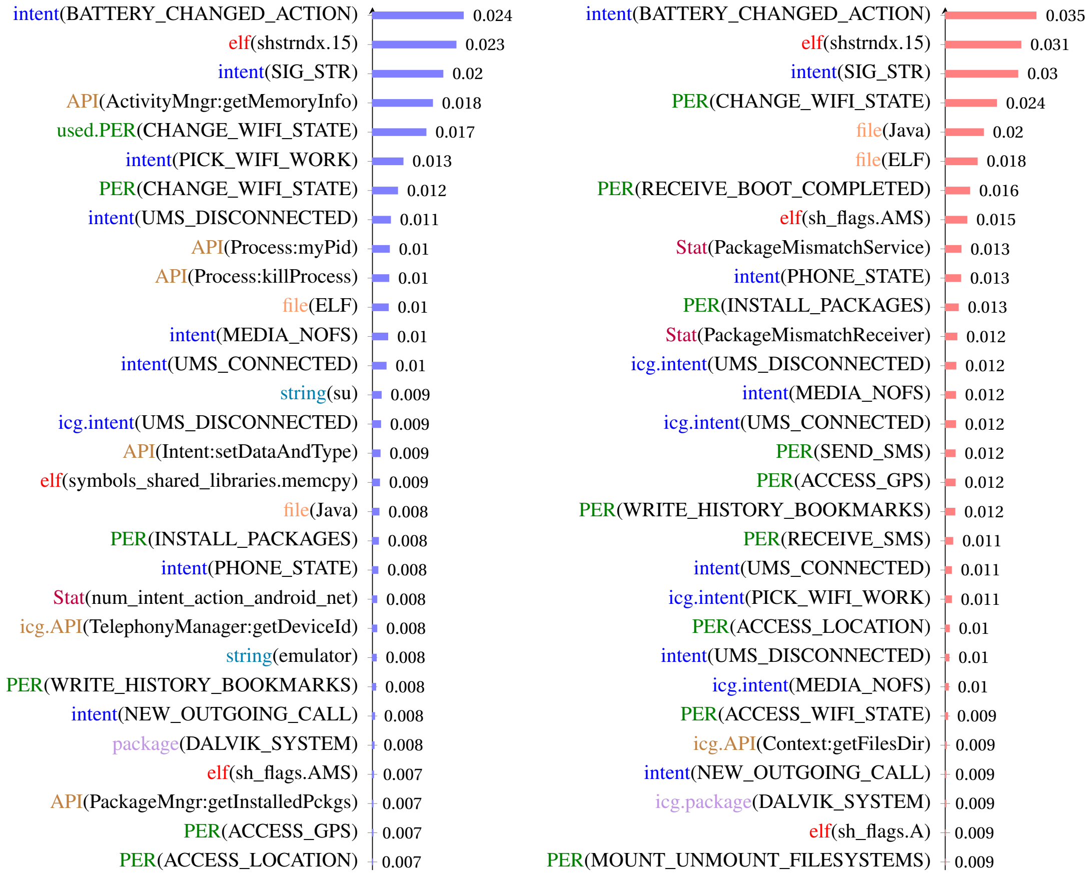
(b) Obfuscated malware.

Figure 5.3: Ranking of features for malware detection: Figure 5.3a shows importance of features by considering all features on MalGenome while Figure 5.3b shows importance of features for the MalGenome obfuscated (PRAGuard) dataset.

programming model. Alongside permissions, novel syntactic features such as used-permissions (prepended with used.PER) also rank highly. These features derived after scanning the code to understand what permissions are being used and how often.

Apart from syntactic features, there are many resource-centric features which also rank highly. In particular, features derived from assets such as ELF files (prepended with elf) as well as intents, and API calls from incognito apps (prepended with icg) rank highly when detecting plain malware samples as shown in Figure 5.3a.

The high-ranked features for malware detection is similar for both plain and ob-



(a) Non-obfuscated malware.

(b) Obfuscated malware.

Figure 5.4: Ranking of features for family identification.

fuscated apps. A noticeable difference in the case of obfuscated malware is that the top-ranked feature is Stat(cert_diff.1). which is derived from the certificate of the app. It checks whether the time difference between the date when the certificate was issued and time when the app was signed is within a day. A temporal proximity means that the app was signed during a time when the malware developer piggybacked the app with malicious code. This is a common practice which signals that the malware developer may be using automated tools to repackage the app.

The ranking of features for classifying malware into families for plain and obfuscated malware is shown in Figures 5.4a and 5.4b, respectively. The high-ranked features in both cases are similar to those observed in the case of classification except for two noticeable differences. Firstly, incognito features are not as important for classifying malware into families as they are for malware detection. This is understandable as incognito apps are a means to achieve a malicious action but they do not characterize what malicious action is carried out or how it is carried out. Secondly, we can see that features derived from the file type of the assets (prepended with file) and those related to logical inconsistencies (features such as Stat(PackageMismatchService) and Stat(PackageMismatchReceiver)) are highly ranked. This could point to the fact that the app is repackaged using an attack vector that is specific to a given family.

5.4.3 Classification Results

In this section we evaluate the effectiveness of DroidSieve in classifying unobfuscated malware, to allow a comparison against approaches from the literature. To not put DroidSieve at a disadvantage, we therefore start with a feature set that includes all features, including those that are susceptible to obfuscation.

As datasets, We first evaluate on detection of malware samples where we use the dataset obtained by combining malicious samples from the Drebin dataset with the Goodware set as shown in Table 5.2b. Note that we only report results for the Drebin dataset here because it includes all MalGenome samples and is both larger and more recent.

Malware Detection. The table shows that in our best scenario we are able to identify if a given app is malicious or benign with accuracy of 99.64% for the case of Drebin, and 99.82% for MvGW. The breakdown of the accuracy shows a detection rate of 99.44% for Drebin, with 0.226% of false positives. Similarly, the detection rate for MvGW is 98.42% with only 0.008% of false positives. For the case of Drebin we obtained slightly higher detection rate with respect to MvGW. However, the false positive rate is better in the case of MvGW. In fact, in this case the number of goodware classified as malware is negligible (2 out of 25493). In most cases, the performance is improved with feature

Type	Classifier	#F	ACC(%)	F1(%)	DR(%)	FPR(%)
Malware Detection		**Drebin + McGW**				
	Extra Trees	22,584	**99.64**	**99.64**	**99.44**	**0.226**
	Extra Trees + FS	**859**	99.57	99.57	99.39	0.302
		MvGW + MvMW				
	Extra Trees	26,396	99.72	99.72	97.58	0.012
	Extra Trees + FS	**634**	**99.82**	**99.81**	**98.42**	**0.008**
Family Identification		**Drebin** (108 families)				
	Extra Trees	2,564	97.68	97.31	–	–
	Extra Trees + FS	**320**	**98.12**	**97.84**	–	–

Table 5.3: Results for detection and family classification on unobfuscated malware with and without Feature Selection (FS) for the Marvin, McAfee and Drebin datasets. *#F* stands for number of features, *ACC* for Accuracy, *F1* for F_1-Score, *DR* for the detection rate, and *FPR* for False Positive Rate. Best scores for each setting are shown in bold. Although feature selection drastically reduces the number of features, it mostly outperforms the full-feature setting.

selection. It allows to drastically reduce the complexity of the feature space, e.g., from over 20,000 features to less than 1,000. This means that we are able to reduce redundant or irrelevant features and improve the performance of classification.

Family Identification. After detection, DroidSieve is also able to determine if the given malware belongs to a known family. Our experiments on the Drebin dataset show that Extra Trees achieve an accuracy of 97.68% when considering all 2,564 the features (see Table 5.2b). Interestingly, keeping the top 320 most informative features increases the accuracy to 98.12% while adding features that are not unimportant can hurt classification accuracy [224].

5.4.4 Obfuscation Evaluation

We now evaluate the effectiveness of our system against obfuscated malware and against a mix of obfuscated and unobfuscated malware, as it would be encountered in an actual deployment. In particular, we ran three sets of experiments for both malware detection and family identification. The three cases are based on scenarios where the training and/or testing samples are obfuscated. Note that our original dataset consists of samples from the Goodware set and samples from the MalGenome project. For each malware sample, we obtain the corresponding obfuscated sample from the PRAGuard project.

Detection of Obfuscated Malware. Our training sets for malware detection are as follows:

1. **McGW + MgGW**: We train on samples from the Goodware and MalGenome data sets only to show a baseline classification without obfuscation.

2. **McGW + PgMW**: We train on the obfuscated malware samples from PRAGuard and include the Goodware.

3. **McGW + MgMW + PgMW**: We train on both the original and obfuscated versions of the malware obtained from MalGenome and PRAGuard, respectively, together with the Goodware.

We chose our test cases for the trained model to highlight that the choice of our features performs consistently well regardless of whether we train on the obfuscated samples or on the original ones. In the first experiment on detecting malware, we retained 33% of samples from PgMW and McGW and trained with the rest. With the retained samples, we obtained accuracies of 100% for the McGW samples (0% false positives) and 99.02% for the PgMW samples. We repeated the experiment with the non-obfuscated set of samples (MgGW + McGW) and obtained similar accuracy values.

To further validate our features and trained models, we also tested on malware samples from a dataset that is different from the one used for training (i.e., 100% hold-out). First, we trained on all MgMW + McGW samples, and tested on PgMW samples. Then, we trained on all PgMW + McGW samples, and tested on MgMW samples. For these two experiments, our accuracy was 92.38% and 96.11% respectively. As a final experiment to validate our features for detection, we also performed a hold-out validation of the 33% of the dataset on all samples i.e. McGW + MgMW + PgMW and obtained an accuracy of 99.71%. A summary of our results for the detection task can be found in Table 5.4. These results show that our features are effective at distinguishing benign and malicious samples, a task made more difficult by many obfuscation techniques also having valid use cases in benign software (see Table 5.1).

To compare our performance with recent approaches, we use Drebin framework [48] to extract features from MgMW, McGW and PgMW datasets. We trained on all MgMW + McGW samples and tested on the obfuscated set of samples (i.e.: PgMW) using the same classification algorithm (Random Forest) used by DroidSieve. The detection rate obtained with Drebin's feature engineering is 0%. Note that our framework reported 92.38% on this experiment. We repeated the same experiment by training over the original set of malware samples collected by Drebin and testing again on PgMW. The feature set of Drebin in this setting is of 101,055 features while ours is of 22,584. After applying feature selection, Drebin retained 13,602 while we retained 859 informative features. For this experiment, the features used by the Drebin framework reported a detection rate of 11% while our framework reported 100%. Among the most important features used by Drebin were different strings such as URLs which are a soft target for obfuscation. Contrarily, our framework retained several *logical inconsistencies* (e.g.: PackageMissmatchIntentConsts and PackageMissmatchService), other resource-centric features (e.g.: PackageNameEntropy), ELF features and other statistical features (such as the number of third party permissions found).

Identification of Obfuscated Families. We now demonstrate the effectiveness of our features for identifying the classes each malware sample belongs to. Our training sets for the identification of malware families are as follows:

1. **MgMW**: We train on samples from MalGenome only.

2. **PgMW**: We train on the obfuscated malware samples from PRAGuard.

3. **MgMW + PgMW**: We train on both the original and obfuscated versions of the malware obtained from MalGenome and PRAGuard respectively.

Malware Detection				**Family Identification**		
	Test				**Test**	
Training	McGW	MgMW	_PgMW_	**Training**	MgMW	_PgMW_
MgMW + McGW	100.00	99.02	_92.38*_	MgMW	97.79	_97.94*_
PgMW + McGW	100.00	96.11*	_99.02_	_PgMW_	97.86*	_99.26_
MgMW + PgGW + McGW	**_99.71_**			MgMW + _PgMW_	**_99.15_**	

Table 5.4: Evaluation of classification on the *McAfee Goodware* (McGW), *Malgenome* (MgMW), and *PRAGuard* (Malgenome obfuscated–*PgMG*) dataset with feature filtering and using hold-out validation (*100% hold-out ratio, otherwise we use the hold-out ration described in Table 5.2b).

By following the same settings as in the previous experiments, for each dataset we retained 33% of the samples from each family, when that dataset was used for both training and testing. A summary of our results on family identification can also be found in Table 5.4. The accuracy after training on MgMW samples was 97.79% and the accuracy after training on PgMW was 99.26%. Additionally, we also applied 100% hold-out validation between MgMW and PgMW showing accuracies of 97.94% and 97.86% respectively. It is worth noting here that training on obfuscated malware enables our classifier to perform better. On the contrary, when obfuscated samples are not included in the training set, the resulting model is not able to prioritize all features needed to perform higher than 97.79%. Finally, we tested the trained models on a combination of both obfuscated and non-obfuscated samples (MgMW + PgMW) and obtained an overall accuracy of 99.15%.

5.4.5 Efficiency

A main design point for DroidSieve was to allow computationally inexpensive feature extraction. Figure 5.5 shows the runtimes for feature extraction on the Marvin dataset, which contains more than 100,000 samples. The median lies at just 2.53 seconds for processing one sample on a single core (Intel Xeon E5-2697 v3 @ 2.60GHz). The overall time for feature extraction on the Marvin dataset took less than 8 hours when executed in parallel on 40 cores.

Other approaches that have been proposed and shown effective for obfuscation-resilient Android malware detection are based on analyzing information flow [53, 129]. However, information flow analysis requires running times that are several orders of magnitude above those seen in DroidSieve's feature extraction. In particular, when attempting to process the 5,560 samples of the Drebin dataset with FlowDroid [49], we were only able to finish half the dataset within three days. Hence, we believe DroidSieve to be better suited for deployment of obfuscation-resilient detection at scale.

5.5 Comparison with Related Works

We now give an overview of the most relevant Android malware detection and classification techniques (see Table 5.5 for a summary), and compare their characteristics with our system.

Malware Detection. Several systems perform Android malware detection, i.e., perform binary classification [36, 48]. DroidAPIMiner [36] is a detection system based on features generated at API level. Drebin [48] is a lightweight detection method that uses static analysis to gather the most important characteristics of Android applications such as permissions, API calls, and network addresses declared in clear text. It uses machine learning (Support Vector Machines) to detect whether a given sample is malicious or benign. DroidSIFT [280] builds contextual API dependency graphs that provide an abstracted view of the possible behaviors of malware and employs machine learning and

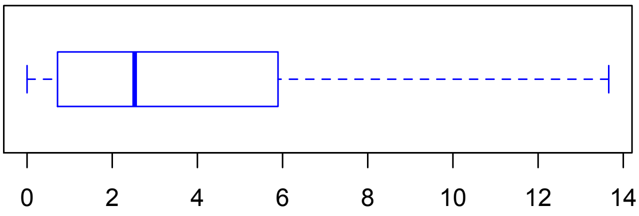


Figure 5.5: Frequency distribution of running times for feature extraction, in seconds. Most samples require less than six seconds to be analyzed.

Year	Method	Type		Feature	# Malware	DR/FP(%)	ACC(%)	Time(s)	Environment
		Det	**Class**						
2014	Dendroid [243]	–	✓	CFG	1,247	–	94	–	
2014	DroidAPIMiner [36]	✓	–	API,PKG,PAR	3,987	99/2.2	–	15	Core i5,6G RAM
2014	DroidMiner [275]	✓	✓	CG,API	2,466	95.3/0.4	92	19.8	–
2014	Drebin [48]	✓	–	PER,STR,API,INT	5,560	94.0/1.0	–	0.75	Core 2 Duo, 4G RAM
2014	DroidSIFT [280]	✓	✓	API-F	2,200	98.0/5.15	93	175.8	Xeon, 128G RAM
2014	DroidLegacy [101]	✓	✓	API	1,052	93.0/3.0	98	–	–
2015	AppAudit [267]	✓	–	API-F	1,005	99.3/0.61	–	0.6	Core i7, 8G RAM
2015	MudFlow [53]	✓	–	API-F	10,552	90.1/18.7	–	–	–
2015	Marvin [171]	✓	–	PER, INT, ST, PN	15,741	98.24/0.0	–	–	–
2015	RevealDroid [129]	✓	✓	PER,API,API-F,INT,PKG	9,054	98.2/18.7	93	95.2	8-Core, 64G RAM
2016	DroidScribe [88]	–	✓	SYSC, BIND, FILE, NET	5,246	–	94	–	–
2016	Madam [227]	✓	–	SYSC, API, PER, SMS, USR	2,800	96/0.2	–	–	–
Ours	**DroidSieve**	✓	✓	*As described in §5.3.1*	16,141	**99.3/0.0**	**99**	2.5	40-Core Xeon, 378G RAM

API: Application Programming Interface, API-F: Information Flow between APIs, INT: Intents, CG: Call Graph, PER: Requested Permissions, CFG: Control Flow Graph, STR: Embedded strings, PKG: Package information of API, ST: Statistical features, PN: Package names, SYSC: System calls, BIND: Binder transactions, FILE: Filesystem Transactions, NET: Network Transactions, USR: User Activity, SMS: SMS Monitoring

Table 5.5: Static analysis techniques on Android malware. Results are reported based on the most representative setting. (Almost all of the systems have difficulty against reflection as they are mostly based on API). The performance time of different systems is subjected to specification of computing environments.

graph similarity to detect malicious applications. MudFlow [53] and AppAudit [267], Ahmadi et al. [37], however, leverage the analysis of flows between APIs to detect malware.

The main weakness of semantics-based static analysis is that it generally shows poor performance against encryption, reflection, native code, and other cross-platform code such as HTML5. These drawbacks motivate dynamic analysis and hybrid approaches [171, 227]. Marvin [171] shows how the combination of static and dynamic analysis can improve the detection rate as well as reducing the number of false positives. It uses a number of statically extracted features and combines them with additional dynamically extracted features, overall more than 490,000. Moreover, it leverages machine learning to detect malware as well as providing a risk score associated with a given unknown sample. Madam [227] proposed a host-based malware detection system that analyzed features at four levels: kernel, application, user and package. It derived features such as system calls, sensitive API calls and SMS through dynamic analysis while complementing these with statically derived features such as permissions, the app's metadata and market information.

Malware Family Classification. In addition to malware detection systems, a number of methods have been proposed just for classification [88, 243] and others [275, 280] evaluated the features used by their detection system for classification. DroidLegacy [101] is a system using API signature similarity to detect and classify malware. Dendroid [243] proposed an approach based on text mining to automatically classify malware samples and analyze families based on the control flow structures found in them. Similarly, RevealDroid [129] aims at identifying Android malware families. Their approach uses information flow analysis and sensitive API flow tracking built on top of two machine learning classifiers, i.e., C4.5 and 1NN. DroidScribe [88] used a purely dynamic approach to malware classification and classified malware into families by observing system calls, Android ICC through the Binder protocol and file/network transactions made by the app. To classify apps that could not be satisfactorily stimulated during dynamic analysis, DroidScribe built on a statistical evaluation framework of the underline machine learning approach [159] to properly trigger a set-based classification scheme that identified the top matching families for a malware sample, given a desired statistical confidence level.

Discussion. We summarize the most prominent static analysis approaches for Android malware analysis tailored to either detection or classification in Table 5.5. The column `Type` shows whether a system was mainly proposed for detection or classification. The `Feature` column shows the extracted attributes from malware. `# Malware` is the total number of malware considered for evaluation. `DR/FP` refers to the detection rate and false positive rate of a detection system, and `ACC` shows the accuracy of the system when it is applied for malware family classification. `Time` shows the average required time for analysis of every application.

Systems like DroidMiner, DroidAPIMiner and Drebin are mainly based on APIs, which are inherently vulnerable to reflection. API-flow based approaches like RevealDroid, AppAudit, MudFlow, and DroidSIFT are more precise and consider features related to the semantics of application, but they are still vulnerable to reflection. Furthermore, flow extraction is expensive unless done in the manner of AppAudit where efficiency is derived from incomplete flow coverage.

In contrast, our system is robust against obfuscation techniques like reflection and encryption while still being computationally efficient. Additionally, while past studies focus on a smaller set of behaviors, our method encompasses a larger set of characteristics and behaviors to distinguish goodware from malware and to identify Android malware families more effectively.

Finally, Roy et al. [224] discuss design choices for evaluating detection systems. Going forward, we plan on taking their important lessons into account. As of now, the focus in DroidSieve lies decidedly on comparing our novel feature engineering for potentially obfuscated malware against existing results in their published settings.

5.6 Conclusion

In this chapter, we have presented a fast, scalable, and accurate system for Android malware detection and family identification based on lightweight static analysis. DroidSieve uses deep inspection of Android malware to build effective and robust features suitable for computational learning. This is key in scenarios where security analysts require intelligent instruments to automate detection and further analysis of Android malware.

We have introduced a novel set of characteristics and showed the importance of systematic feature engineering to achieve a diversified and large range of features that can adjust to different malware. Our findings show that static analysis for Android can succeed even when confronted with obfuscation techniques such as reflection, encryption and dynamically-loaded native code. While fundamental changes in characteristics of malware remain a largely open problem, we showed that DroidSieve remains resilient against state-of-the-art obfuscation techniques which can be used to quickly derive new and syntactically different malware variants.

Chapter 6

Concluding Remarks

In this thesis, we focused on modeling the functions of Android applications that can be misused by attackers and the security community has neglected addressing that attributes of applications. Hence, we targeted three different parts of Android applications and showed how they can help to improve the effectiveness of an Android classification system. First, we proposed a detection system based on extracting signatures from HTTP traffic of Android applications and represented the effectiveness of the system compared to when the approach is used for desktop programs. Second, due to the affect of encryption on HTTP traffic by auxiliary libraries, the first approach approach might not be effectively apply to all kinds of Applications like those applications that use GCM to communicate with servers. So we proposed a static approach to better detect those applications that misuse GCM channel. Third, as static analysis approaches can be misled by obfuscation techniques, we addressed modeling obfuscation behaviors of applications to improve the influence of classification systems. All of the proposed techniques in this thesis aim to show how the effective feature engineering based on an understanding of applications functions, which are potential to be exploited by adversary, can help the improvement of learning-based systems.

Bibliography

[1] Enisa threat taxonomy. http://goo.gl/ATLpcA.

[2] Mobile advertisement platforms. http://www.mobyaffiliates.com/mobile-advertising-networks/.

[3] Droid2, 2010.

[4] android-apktool: A tool for reengineering android apk files. https://code.google.com/p/android-apktool/, 2010–2014.

[5] adb trickery #2, 2011.

[6] Android reverse engineering (a.r.e.) virtual machine. https://www.honeynet.org/node/783, 2011.

[7] Keychain. http://developer.android.com/reference/android/security/KeyChain.html, 2011.

[8] Root for android, 2011.

[9] yummy yummy, gingerbreak!, 2011.

[10] Zimperlich sources, 2011.

[11] androguard - reverse engineering, malware and goodware analysis of android applications ... and more. https://code.google.com/p/androguard/, 2011–2013.

[12] Dare: Dalvik retargeting. http://siis.cse.psu.edu/dare/, 2012.

[13] Droidscope. https://code.google.com/p/decaf-platform/wiki/DroidScope, 2012.

[14] copperdroid. http://copperdroid.isg.rhul.ac.uk/copperdroid/, 2013.

[15] dex2jar: Tools to work with android .dex and java .class files. https://code.google.com/p/dex2jar/, 2013.

[16] Root exploits, August 2013.

[17] Soot - a java optimization framework. https://github.com/StevenArzt/soot, 2013.

[18] A5 online service. http://dogo.ece.cmu.edu/a5/, 2014.

[19] A5 source code. https://github.com/tvidas/a5, 2014.

[20] Android security overview. https://source.android.com/devices/tech/security/, 2014.

[21] anubis - malware analysis for unknown binaries. https://anubis.iseclab.org/, 2014.

[22] Dexguard | guardsquare. http://www.saikoa.com/dexguard, 2014.

[23] Dexter. http://dexter.dexlabs.org/, 2014.

[24] droidbox - android application sandbox. https://code.google.com/p/droidbox/, 2014.

[25] Flowdroid - secure software engineering. https://github.com/secure-software-engineering/soot-infoflow-android/wiki, 2014.

[26] jnetpcap opensource | protocol analysis sdk. http://jnetpcap.com/, 2014.

[27] Mobile sandbox (ng). http://mobilesandbox.org/, 2014.

[28] Novel active learning methods for enhanced {PC} malware detection in windows {OS}. *Expert Systems with Applications*, 41(13):5843 – 5857, 2014.

[29] Qemu: open source processor emulator. http://wiki.qemu.org/Main_Page, 2014.

[30] smali - an assembler/disassembler for android's dex format. https://code.google.com/p/smali/, 2014.

[31] Snort - open source network intrusion prevention system. https://www.snort.org, 2014.

[32] Taintdroid - realtime privacy monitoring on smartphones. http://appanalysis.org/index.html, 2014.

[33] Tracedroid - free online dynamic android app nalysis. http://tracedroid.few.vu.nl, 2014.

[34] Uiautomator - android framework. http://developer.android.com/tools/help/uiautomator/index.html, 2014.

[35] Virustotal - free online virus, malware and url scanner. https://www.virustotal.com/, 2014.

[36] Yousra Aafer, Wenliang Du, and Heng Yin. *DroidAPIMiner: Mining API-Level Features for Robust Malware Detection in Android*, pages 86–103. 2013.

[37] Mansour Ahmadi, Battista Biggio, Steven Arzt, Davide Ariu, and Giorgio Giacinto. Detecting misuse of google cloud messaging in android badware. In *Proceedings of the 6th Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '16, pages 103–112, New York, NY, USA, 2016. ACM.

[38] Mansour Ahmadi, Ashkan Sami, Hossein Rahimi, and Babak Yadegari. Malware detection by behavioural sequential patterns. *Computer Fraud & Security*, 2013(8):11 – 19, 2013.

[39] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. Novel feature extraction, selection and fusion for effective malware family classification. In *Conference on Data and Application Security and Privacy (CODASPY)*, pages 183–194, 2016.

[40] Ron Amadeo. App ops: Android 4.3's hidden app permission manager, control permissions for individual apps!, August 2013.

[41] Brandon Amos, Hamilton Turner, and Jules White. Applying machine learning classifiers to dynamic android malware detection at scale. In *Wireless Communications and Mobile Computing Conference (IWCMC), 2013 9th International*, pages 1666–1671, July 2013.

[42] Android. Proguard, 2014.

[43] AndroTotal. (another) android trojan scheme using google cloud messaging. http://blog.andrototal.org/post/89637972097/another-android-trojan-scheme-using-google-cloud, jun 2014.

[44] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 173–187, New York, NY, USA, 2011. ACM.

[45] Axelle Apvrille and Ruchna Nigam. Obfuscation in android malware, and how to fight back. *Virus Bulletin*, pages 1–10, 2014.

[46] Marco Aresu, Davide Ariu, Mansour Ahmadi, Davide Maiorca, and Giorgio Giacinto. Clustering android malware families by http traffic. In *Malicious and Unwanted Software (MALWARE)*, pages 128–135, Oct 2015.

[47] Anshul Arora, Shree Garg, and Sateesh K. Peddoju. Malware detection using network traffic analysis in android based mobile devices. In *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2014 Eighth International Conference on*, pages 66–71, Sept 2014.

[48] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, and Siemens. Drebin: Efficient and explainable detection of android malware in your pocket. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2014.

[49] 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.

[50] N. Asokan, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Kari Kostiainen, Elena Reshetova, and Ahmad-Reza Sadeghi. *Mobile Platform Security*, volume 4 of *Synthesis Lectures on Information Security, Privacy, and Trust*. Morgan & Claypool, December 2013.

[51] Japan Smartphone Security Association. *Android Application Secure Design/Secure Coding Guidebook*. June 2014.

[52] 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, 2012.

[53] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *37th International Conference on Software Engineering (ICSE)*, 2015.

[54] Adam J. Aviv, Katherine Gibson, Evan Mossop, Matt Blaze, and Jonathan M. Smith. Smudge attacks on smartphone touch screens. In *Proceedings of the 4th USENIX Conference on Offensive Technologies*, WOOT'10, pages 1–7, Berkeley, CA, USA, 2010. USENIX Association.

[55] Adam J. Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M. Smith. Practicality of accelerometer side channels on smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 41–50, New York, NY, USA, 2012. ACM.

[56] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. Appguard - enforcing user requirements on android apps. In Nir Piterman and ScottA. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 7795 of *Lecture Notes in Computer Science*, pages 543–548. Springer Berlin Heidelberg, 2013.

[57] 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.

[58] 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.

[59] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. Mockdroid: Trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 49–54, New York, NY, USA, 2011. ACM.

[60] Battista Biggio, Giorgio Fumera, and Fabio Roli. Multiple classifier systems for robust classifier design in adversarial environments. *Int. Journal of Machine Learning and Cybernetics*, 1(1):27–41, 2010.

[61] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, 1 edition, October 2007.

[62] Bitdefender. Clueful detects vulnerable applovin/vulna apps, July 2013.

[63] Thomas Blasing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 55–62, Oct 2010.

[64] bluebox. Android master key exploit - uncovering android master key, July 2013.

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

[66] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *19th Annual Network & Distributed System Security Symposium (NDSS'12)*, Feb 2012.

[67] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 51–62, New York, NY, USA, 2011. ACM.

[68] Sven Bugiel, Stephen Heuser, and Ahmad-Reza Sadeghi. Flexible and fine-grained mandatory access control on android for diverse security and privacy policies. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 131–146, Washington, D.C., 2013. USENIX.

[69] 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.

[70] businessinsider. A scary graphic for android users, August 2014.

[71] Liang Cai and Hao Chen. Touchlogger: Inferring keystrokes on touch screen from smartphone motion. In *Proceedings of the 6th USENIX Conference on Hot Topics in Security*, HotSec'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[72] Liang Cai and Hao Chen. On the practicality of motion based keystroke inference attack. In *Proceedings of the 5th International Conference on Trust and Trustworthy Computing*, TRUST'12, pages 273–290, Berlin, Heidelberg, 2012. Springer-Verlag.

[73] Liang Cai, Sridhar Machiraju, and Hao Chen. Defending against sensor-sniffing attacks on mobile phones. In *Proceedings of the 1st ACM Workshop on Networking, Systems, and Applications for Mobile Handhelds*, MobiHeld '09, pages 31–36, New York, NY, USA, 2009. ACM.

[74] Zhenquan Cai and Roland H.C. Yap. Inferring the detection logic and evaluating the effectiveness of android anti-virus apps. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 172–182, 2016.

[75] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 13–24, New York, NY, USA, 2013. ACM.

[76] Qi Alfred Chen, Zhiyun Qian, and Z. Morley Mao. Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1037–1052, San Diego, CA, August 2014. USENIX Association.

[77] Yangyi Chen, Tongxin Li, XiaoFeng Wang, Kai Chen, and Xinhui Han. Perplexed messengers from the cloud: Automated security analysis of push-messaging integrations. In *Computer and Communications Security (CCS)*, pages 1260–1272, 2015.

[78] 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.

[79] CIO. X-ray app identifies android vulnerabilities but doesn't fix them, August 2012.

[80] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical Report TR148, Department of Computer Science, University of Auckland, 1997.

[81] Mauro Conti, Luigi V. Mancini, Riccardo Spolaor, and Nino Vincenzo Verde. Can't you hear me knocking: Identification of user actions on android apps via traffic analysis. *Fifth ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2015.

[82] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. Crepe: Context-related policy enforcement for android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 331–345, Berlin, Heidelberg, 2011. Springer-Verlag.

[83] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: attacks and defenses for the vulnerability of the decade. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 2, pages 119–129 vol.2, 2000.

[84] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the clones: Detecting cloned applications on android markets. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security - ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 37–54. Springer Berlin Heidelberg, 2012.

[85] CVE. Cve-2011-1717, May 2011.

[86] CVE. Android vulnerability statistics, November 2016.

[87] Shuaifu Dai, Alok Tongaonkar, Xiaoyin Wang, Antonio Nucci, and Dawn Song. Networkprofiler: Towards automatic fingerprinting of android apps. In *INFOCOM*, pages 809–817. IEEE, 2013.

[88] Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, Mansour Ahmadi, Johannes Kinder, and Lorenzo Cavallaro. Droidscribe: Classifying android malware based on runtime behavior. In *Mobile Security Technologies (MoST)*, 2016.

[89] National Vulnerability Database. Cve-2009-2692, August 2009.

[90] National Vulnerability Database. Cve-2010-1185, March 2010.

[91] National Vulnerability Database. Cve-2011-1149, March 2011.

[92] National Vulnerability Database. Cve-2011-1350, March 2011.

[93] National Vulnerability Database. Cve-2011-1823, April 2011.

[94] National Vulnerability Database. Cve-2011-3874, October 2011.

[95] National Vulnerability Database. Cve-2012-0056, January 2012.

[96] National Vulnerability Database. Cve-2014-3153, June 2014.

[97] National Vulnerability Database. Cve-2015-3636, May 2015.

[98] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th International Conference on Information Security*, ISC'10, pages 346–360, Berlin, Heidelberg, 2011. Springer-Verlag.

[99] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 559–570, New York, NY, USA, 2013. ACM.

[100] Amit Deo, Santanu Kumar Dash, Guillermo Suarez-Tangil, Vladimir Vovk, and Lorenzo Cavallaro. Prescience: Probabilistic guidance on the retraining conundrum for malware detection. In *ACM Workshop on Artificial Intelligence and Security (AISec)*, 2016.

[101] Luke Deshotels, Vivek Notani, and Arun Lakhotia. DroidLegacy: Automated familial classification of Android malware. In *ACM SIGPLAN on Program Protection and Reverse Engineering Workshop (PPREW)*, 2014.

[102] Revolutionary dev team. zergrush android 2.2 / 2.3 local root, 2011.

[103] Tom Dietterich. Overfitting and undercomputing in machine learning. *ACM Computing Surveys (CSUR)*, 27(3):326–327, September 1995.

[104] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 23–23, Berkeley, CA, USA, 2011. USENIX Association.

[105] Jason A. Donenfeld. Mempodroid exploit, January 2012.

[106] William Enck. Defending users against smartphone apps: Techniques and future directions. In *Proceedings of the 7th International Conference on Information Systems Security*, ICISS'11, pages 49–70, Berlin, Heidelberg, 2011. Springer-Verlag.

[107] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference*

on Operating Systems Design and Implementation, OSDI'10, pages 1–6, Berkeley, CA, USA, 2010. USENIX Association.

[108] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA, 2011.

[109] William Enck, Machigar Ongtang, and Patrick Mcdaniel. Mitigating android software misuse before it happens. Technical report, The Pennsylvania State University, 2008.

[110] 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*, CCS'09, pages 235–245, New York, NY, USA, 2009. ACM.

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

[112] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Ithaca, NY, USA, 2004. AAI3114521.

[113] exploit db. Linux kernel 2.x - sock_sendpage() local root exploit (android edition), August 2009.

[114] exploit db. Android 1.x/2.x - local root exploit, July 2010.

[115] f secure. Mobile threat report q1 2014, 2014.

[116] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. Why eve and mallory love android: An analysis of android ssl (in)security. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 50–61, New York, NY, USA, 2012. ACM.

[117] Sascha Fahl, Marian Harbach, Marten Oltrogge, Thomas Muders, and Matthew Smith. Hey, you, get off of my clipboard. In Ahmad-Reza Sadeghi, editor, *Financial Cryptography and Data Security*, volume 7859 of *Lecture Notes in Computer Science*, pages 144–161. Springer Berlin Heidelberg, 2013.

[118] Zheran Fang, Weili Han, and Yingjiu Li. Permission based android security: Issues and countermeasures. *Computers & Security*, 43(0):205 – 218, 2014.

[119] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android security: A survey of issues, malware penetration and defenses. *Communications Surveys Tutorials, IEEE*, PP(99):1–1, 2015.

[120] 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.

[121] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Computer and Communications Security (CCS)*, pages 627–638, 2011.

[122] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 3–14, New York, NY, USA, 2011. ACM.

[123] Adrienne Porter Felt and David Wagner. Phishing on mobile devices. In *In W2SP*, 2011.

[124] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steven Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association.

[125] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *The Journal of Machine Learning Research (JMLR)*, 15(1):3133–3181, January 2014.

[126] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. Modeling and enhancing android's permission system. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security - ESORICS 2012*, volume 7459 of *Lecture Notes in Computer Science*, pages 1–18. Springer Berlin Heidelberg, 2012.

[127] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. TriggerScope: Towards Detecting Logic Bombs in Android Apps. In *Security and Privacy (SP)*, May 2016.

[128] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. Scandroid: Automated security certification of android applications. Technical Report CS-TR-4991, Department of Computer Science, University of Maryland, College Park, November 2009.

[129] Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. Technical report, Dept. of Computer Science, George Mason University, 2015.

[130] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006.

[131] Giorgio Giacinto, Fabio Roli, and Luca Didaci. Fusion of multiple classifiers for intrusion detection in computer networks. *Pattern Recogn. Lett.*, 24(12):1795–1803, August 2003.

[132] 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.

[133] Google. Android and security, February 2012.

[134] Google. Android device manager, August 2013.

[135] Google. Seforandroid, July 2013.

[136] Google. Multiple external storage devices, January 2014.

[137] Google. Program policies google play for developers, June 2014.

[138] Google. Security enhancements in android 5.0, October 2014.

[139] Google. Creating better user experiences on google play, March 2015.

[140] 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.

[141] 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.

[142] Michael C. Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. Systematic detection of capability leaks in stock android smartphones. In *19th Annual Network and Distributed System Security Symposium, NDSS*, San Diego, California, USA, February 2012. The Internet Society.

[143] Boxuan Gu, Xinfeng Li, Gang Li, A.C. Champion, Zhezhe Chen, Feng Qin, and Dong Xuan. D2taint: Differentiated and dynamic information flow tracking on smartphones for numerous data sources. In *INFOCOM, 2013 Proceedings IEEE*, pages 791–799, April 2013.

[144] Jiaping Gui, Stuart Mcilroy, Meiyappan Nagappan, and William G. J. Halfond. Truth in advertising: The hidden cost of mobile ads for software developers. In *Int. Conf. on Software Engineering (ICSE)*, pages 100–110, 2015.

[145] Jun Han, E. Owusu, L.T. Nguyen, A. Perrig, and J. Zhang. Accomplice: Location inference using accelerometers on smartphones. In *Communication Systems and Networks (COMSNETS), 2012 Fourth International Conference on*, pages 1–9, Jan 2012.

[146] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, oct 1988.

[147] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference, and prediction: with 200 full-color illustrations*. New York: Springer-Verlag, 2 edition, 2009.

[148] Tsung-Hsuan Ho, Daniel Dean, Xiaohui Gu, and William Enck. Prec: Practical root exploit containment for android devices. In *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 187–198, New York, NY, USA, 2014. ACM.

[149] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 639–652, New York, NY, USA, 2011. ACM.

[150] George Hotz. Towelroot, June 2014.

[151] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, pages 611–620, New York, NY, USA, 2009. ACM.

[152] IDC. Smartphone os market share, q2 2016, 2016.

[153] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

[154] Suman Jana and Vitaly Shmatikov. Memento: Learning secrets from process footprints. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 143–157, Washington, DC, USA, 2012. IEEE Computer Society.

[155] 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*, SPSM '12, pages 3–14, New York, NY, USA, 2012. ACM.

[156] Xuxian Jiang. An evaluation of the application (app) verification service in android 4.2, 2012.

[157] Yiming Jing, Ziming Zhao, Gail-Joon Ahn, and Hongxin Hu. Morpheus: Automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference*, ACSAC '14, pages 216–225, New York, NY, USA, 2014. ACM.

[158] Jon Oberheide Jon Larimer. levitator exploit, March 2011.

[159] Roberto Jordaney, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Misleading Metrics: On Evaluating Machine Learning for Malware with Confidence. Technical Report 2016-1, Royal Holloway, University of London, 2016.

[160] Kaspersky. Gcm in malicious attachments. https://securelist.com/blog/mobile/57471/gcm-in-malicious-attachments/, August 2013.

[161] Amin Kharraz, Engin Kirda, William Robertson, Davide Balzarotti, and Aurelien Francillon. Optical delusions: A study of malicious qr codes in the wild. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 192–203, June 2014.

[162] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, SOAP'14, pages 1–6, New York, NY, USA, 2014. ACM.

[163] Ludmila I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. John Wiley and Sons, Inc., second edition, 2014.

[164] Ludmila I. Kuncheva. *Ensemble Methods*, pages 186–229. John Wiley & Sons, Inc., 2014.

[165] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4android: A generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 39–50, New York, NY, USA, 2011. ACM.

[166] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From zygote to morula: Fortifying weakened aslr on android. In *Proceedings of the 2014 IEEE Symposium*

on Security and Privacy, SP '14, pages 424–439, Washington, DC, USA, 2014. IEEE Computer Society.

[167] Li Li, Alexandre Bartel, Tegawende F Bissyande, Jacques Klein, Yves Le Traon, Steven Arzt, Rasthofer Siegfried, Eric Bodden, Damien Octeau, and Patrick Mcdaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering (ICSE 2015)*, 2015.

[168] Li Li, Tegawendé F Bissyandé, Damien Octeau, and Jacques Klein. Reflection-aware static analysis of android apps. In *Automated Software Engineering, Demo Track (ASE)*, 2016.

[169] Tongxin Li, Xiaoyong Zhou, Luyi Xing, Yeonjoon Lee, Muhammad Naveed, XiaoFeng Wang, and Xinhui Han. Mayhem in the push clouds: Understanding and mitigating security hazards in mobile push-messaging services. In *Computer and Communications Security (CCS)*, pages 978–989, 2014.

[170] Chia-Chi Lin, Hongyang Li, Xiaoyong Zhou, and XiaoFeng Wang. Screenmilker: How to milk your android screen for secrets. *Network and Distributed System Security (NDSS) Symposium 2014*, 2014.

[171] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis. In *Proceedings of the 39th Annual International Computers, Software & Applications Conference (COMPSAC)*, volume 2, pages 422–433, July 2015.

[172] 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 *3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[173] Martina Lindorfer, Matthias Neumayr, Juan Caballero, and Christian Platzer. Poster: Cross-platform malware: write once, infect everywhere. In *Proceedings of the 2013 ACM SIGSAC conference on Computer &#38; communications security*, CCS '13, pages 1425–1428, New York, NY, USA, 2013. ACM.

[174] Anthony Lineberry, David Luke Richardson, and Tim Wyatt. These aren't the permissions you are looking for. In *DEF CON 18,*, 2010.

[175] lookout. Lookout mobile threat report, August 2011.

[176] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[177] Gilles Louppe, Louis Wehenkel, Antonio Sutera, and Pierre Geurts. Understanding variable importances in forests of randomized trees. In *Advances in Neural Information Processing Systems (NIPS)*, pages 431–439, 2013.

[178] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: Statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS'12, pages 229–240, New York, NY, USA, 2012. ACM.

[179] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51(0):16 – 31, 2015.

[180] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Capkun. Analysis of the communication between colluding applications on modern smartphones. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 51–60, New York, NY, USA, 2012. ACM.

[181] Yan Michalevsky, Dan Boneh, and Gabi Nakibly. Gyrophone: Recognizing speech from gyroscope signals. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 1053–1067, San Diego, CA, Aug 2014.

[182] Emiliano Miluzzo, Alexander Varshavsky, Suhrid Balakrishnan, and Romit Roy Choudhury. Tapprints: Your finger taps have fingerprints. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, MobiSys '12, pages 323–336, New York, NY, USA, 2012. ACM.

[183] Collin Mulliner, Jon Oberheide, William Robertson, and Engin Kirda. Patchdroid: Scalable third-party security patches for android devices. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 259–268, New York, NY, USA, 2013. ACM.

[184] Adwait Nadkarni and William Enck. Preventing accidental data disclosure in modern operating systems. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, CCS '13, pages 1029–1042, New York, NY, USA, 2013. ACM.

[185] Masoud Narouei, MansourAhmadi, Giorgio Giacinto, Hassan Takabi, and Ashkan Sami. Dllminer: Structural mining for malware detection. *Security and Communication Networks*, 2015.

[186] FairuzAmalina Narudin, Ali Feizollah, NorBadrul Anuar, and Abdullah Gani. Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, pages 1–15, 2014.

[187] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, New York, NY, USA, 2010. ACM.

[188] Netapp. Mobile/tablet top operating system share trend, Oct 2016.

[189] James Newsome, Brad Karp, and Dawn Xiaodong Song. Polygraph: Automatically generating signatures for polymorphic worms. In *2005 IEEE Symposium on Security and Privacy (S&P 2005), 8-11 May 2005, Oakland, CA, USA*, pages 226–241, 2005.

[190] NextApp. Sdfix: Kitkat writable microsd, 2014.

[191] Ruchna Nigam. A timeline of mobile botnets. *Virus Bulletin*, March 2015.

[192] Jon Oberheide and Charlie Miller. Dissecting the android bouncer. In *SummerCon*, Brooklyn, NY, USA, June 2012.

[193] 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.

[194] Damien Octeau, Patrick McDaniel, Somesh Jha, Alexandre Bartel, Eric Bodden, Jacques Klein, and Yves Le Traon. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558, Washington, D.C., 2013. USENIX.

[195] Machigar Ongtang, Kevin Butler, and Patrick McDaniel. Porscha: Policy oriented secure content handling in android. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 221–230, New York, NY, USA, 2010. ACM.

[196] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC'09, pages 340–349, Washington, DC, USA, 2009. IEEE Computer Society.

[197] Emmanuel Owusu, Jun Han, Sauvik Das, Adrian Perrig, and Joy Zhang. Accessory: Password inference using accelerometers on smartphones. In *Proceedings of the Twelfth Workshop on Mobile Computing Systems &#38; Applications*, HotMobile '12, pages 9:1–9:6, New York, NY, USA, 2012. ACM.

[198] Mark Palatucci, Dean Pomerleau, Geoffrey E Hinton, and Tom M Mitchell. Zero-shot learning with semantic output codes. In *Advances in neural information processing systems (NIPS)*, pages 1410–1418, 2009.

[199] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. Whyper: Towards automating risk assessment of mobile applications. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 527–542, Washington, D.C., 2013. USENIX.

[200] Yeongung Park, ChoongHyun Lee, Chanhee Lee, JiHyeog Lim, Sangchul Han, Minkyu Park, and Seong-Je Cho. Rgbdroid: A novel response-based approach to android privilege escalation attacks. In *Proceedings of the 5th USENIX Conference on Large-Scale*

Exploits and Emergent Threats, LEET'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[201] M. Parkour. Contagio mobile - mobile malware mini dump. http:///contagiominidump.blogspot.it/, 2012.

[202] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. Addroid: Privilege separation for applications and advertisers in android. In *Symp. on Information, Computer and Communications Security (ASIACCS)*, pages 71–72, 2012.

[203] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 241–252, New York, NY, USA, 2012. ACM.

[204] Roberto Perdisci, Davide Ariu, Prahlad Fogla, Giorgio Giacinto, and Wenke Lee. Mcpad: A multiple classifier system for accurate payload-based anomaly detection. *Comput. Netw.*, 53(6):864–881, April 2009.

[205] Roberto Perdisci, Davide Ariu, and Giorgio Giacinto. Scalable fine-grained behavioral clustering of http-based malware. *Computer Networks*, 57(2):487–500, 2013.

[206] Roberto Perdisci, Wenke Lee, and Nick Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 26–26, Berkeley, CA, USA, 2010. USENIX Association.

[207] 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*, EuroSec '14, pages 5:1–5:6, New York, NY, USA, 2014. ACM.

[208] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute this! analyzing unsafe and malicious dynamic code loading in android applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2014.

[209] David Poll. Discovering a major security hole in facebook's android sdk, April 2012.

[210] Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra. A survey on security for mobile devices. *Communications Surveys Tutorials, IEEE*, 15(1):446–471, January 2013.

[211] 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.

[212] Irena Prochkova, Varun Singh, and Jukka K. Nurminen. Energy cost of advertisements in mobile games on the android platform. In *Int. Conf. on Next Generation Mobile Applications, Services and Technologies*, pages 147–152, Sept 2012.

[213] Chenxiong Qian, Xiapu Luo, Yuru Shao, and Alvin T. S. Chan. On tracking information flows through jni in android applications. In *Dependable Systems and Networks (DSN)*, pages 180–191, 2014.

[214] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. Autocog: Measuring the description-to-permission fidelity in android applications. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1354–1365, New York, NY, USA, 2014. ACM.

[215] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *Network and Distributed System Security Symposium (NDSS)*, February 2014.

[216] Siegfried Rasthofer, Steven Arzt, Marc Miltenberger, and Eric Bodden. Harvesting runtime values in android applications that feature anti-analysis techniques. *Network and Distributed System Security (NDSS)*, 2016.

[217] 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.

[218] 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 6^{th} European Workshop on System Security (EUROSEC)*, Prague, Czech Republic, April 2013.

[219] Google report. Android security, 2014 year in review, 2015.

[220] reuters. Your medical record is worth more to hackers than your credit card, 2014.

[221] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Dussel, and Pavel Laskov. Learning and classification of malware behavior. In *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA '08, pages 108–125, Berlin, Heidelberg, 2008. Springer-Verlag.

[222] Franziska Roesner and Tadayoshi Kohno. Securing embedded user interfaces: Android and beyond. In *Presented as part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 97–112, Washington, D.C., 2013. USENIX.

[223] Sanae Rosen, Zhiyun Qian, and Z. Morely Mao. Appprofiler: A flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 221–232, New York, NY, USA, 2013. ACM.

[224] Sankardas Roy, Jordan DeLoach, Yuping Li, Nic Herndon, Doina Caragea, Xinming Ou, Venkatesh Prasad Ranganath, Hongmin Li, and Nicolais Guevara. Experimental study with real-world data for android app security analysis using machine learning. In *Proceedings of the 31st Annual Computer Security Applications Conference*, pages 81–90. ACM, 2015.

[225] Ashkan Sami, Babak Yadegari, Hossein Rahimi, Naser Peiravian, Sattar Hashemi, and Ali Hamze. Malware detection based on mining api calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1020–1025, New York, NY, USA, 2010. ACM.

[226] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G. Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231(0):64 – 82, 2013. Data Mining for Information Security.

[227] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, PP(99):1–1, 2016.

[228] 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*, SACMAT '12, pages 13–22, New York, NY, USA, 2012. ACM.

[229] Roman Schlegel, Kehuan Zhang, Xiaoyong Zhou, Mehool Intwala, Apu Kapadia, , and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, page 1733, 2011.

[230] Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Jan Clausen, Kamer A. Yuksel, Osman Kiraz, Seyit A. Camtepe, and Sahin Albayrak. Enhancing security of linux-based android devices. In *15th International Linux Kongress*, Hamburg, Germany, October 2008.

[231] Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Computing Surveys*, 49(1), April 2016.

[232] 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, February 2012.

[233] Asaf Shabtai, Lena Tenenboim-Chekina, Dudu Mimran, Lior Rokach, Bracha Shapira, and Yuval Elovici. Mobile malware detection through analysis of deviations in application network behavior. *Computers & Security*, 43(0):1 – 18, 2014.

[234] M.Zubair Shafiq, S.Momina Tabish, Fauzan Mirza, and Muddassar Farooq. Pe-miner: Mining structural information to detect malicious executables in realtime. In Engin Kirda, Somesh Jha, and Davide Balzarotti, editors, *Recent Advances in Intrusion Detection*, volume 5758 of *Lecture Notes in Computer Science*, pages 121–141. Springer Berlin Heidelberg, 2009.

[235] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. Adsplit: Separating smartphone advertising from applications. In *USENIX Conference on Security Symposium*, pages 28–28, 2012.

[236] Stephen Smalley and Robert Craig. Security enhanced (se) android: Bringing flexible mac to android. In *20th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, California, USA, February 2013.

[237] Charles Smutz and Angelos Stavrou. Malicious pdf detection using metadata and structural features. In *28th Annual Computer Security Applications Conference (ACSAC)*, pages 239–248, New York, NY, USA, 2012. ACM.

[238] Sophos. Security threat report, 2014.

[239] 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.

[240] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. Investigating user privacy in android ad libraries. In *Mobile Security Technologies (MoST)*, 2012.

[241] Guillermo Suarez-Tangil, Juan Tapiador, Flavio Lombardi, and Roberto Di Pietro. Alterdroid: Differential fault analysis of obfuscated smartphone malware. 2016.

[242] Guillermo Suarez-Tangil, Juan E Tapiador, and Pedro Peris-Lopez. Stegomalware: Playing hide and seek with malicious components in smartphone apps. In *10th International Conference on Information Security and Cryptology (Inscrypt)*, pages 496–515. Springer, December 2014.

[243] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Jorge Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(1):1104–1117, 2014.

[244] Sufatrio, Darell J. J. Tan, Tong-Wei Chua, and Vrizlynn L. L. Thing. Securing android: A survey, taxonomy, and challenges. *ACM Comput. Surv.*, 47(4):58:1–58:45, May 2015.

[245] symantec. Windows malware attempts to infect android devices, January 2014.

[246] S. Momina Tabish, M. Zubair Shafiq, and Muddassar Farooq. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, CSI-KDD '09, pages 23–31, New York, NY, USA, 2009. ACM.

[247] Yang Tang, Phillip Ames, Sravan Bhamidipati, Ashish Bijlani, Roxana Geambasu, and Nikhil Sarda. Cleanos: Limiting mobile data exposure with idle eviction. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 77–91, Berkeley, CA, USA, 2012. USENIX Association.

[248] Robert Templeman, Zahid Rahman, David Crandall, and Apu Kapadia. PlaceRaider: Virtual theft in physical spaces with smartphones. In *Proceedings of The 20th Annual Network and Distributed System Security Symposium (NDSS)*, February 2013.

[249] Peter Teufl, Michaela Ferk, Andreas Fitzek, Daniel Hein, Stefan Kraxberger, and Clemens Orthacker. Malware detection by applying knowledge discovery processes to application metadata on the android market (google play). *Security and Communication Networks*, 2013.

[250] Alok Tongaonkar, Shuaifu Dai, Antonio Nucci, and Dawn Song. Understanding mobile app usage patterns using in-app advertisements. In Matthew Roughan and Rocky Chang, editors, *Passive and Active Measurement*, volume 7799 of *Lecture Notes in Computer Science*, pages 63–72. Springer Berlin Heidelberg, 2013.

[251] Trendmicro. Android malware use ssl for evasion. http://blog.trendmicro.com/trendlabs-security-intelligence/android-malware-use-ssl-for-evasion/, Sep 2014.

[252] US-CERT/NIST. Vulnerability summary for cve-2014-1484, february 2014.

[253] Timothy Vidas and Nicolas Christin. Sweetening android lemon markets: Measuring and combating malware in application marketplaces. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 197–208, New York, NY, USA, 2013. ACM.

[254] 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.

[255] Timothy Vidas, Nicolas Christin, and Lorrie Faith Cranor. Curbing Android permission creep. In *Proceedings of the Web 2.0 Security and Privacy 2011 workshop (W2SP 2011)*, Oakland, CA, May 2011.

[256] Timothy Vidas, Jiaqi Tan, Jay Nahata, Chaur Lih Tan, Nicolas Christin, and Patrick Tague. A5: Automated analysis of adversarial android applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones &#38; Mobile Devices*, SPSM '14, pages 39–50, New York, NY, USA, 2014. ACM.

[257] Timothy Vidas, Daniel Votipka, and Nicolas Christin. All your droid are belong to us: A survey of current android attacks. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[258] Nedim Šrndic and Pavel Laskov. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP)*, pages 197–211, 2014.

[259] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. Profiledroid: Multi-layer profiling of android applications. In *Proceedings of the 18th Annual International Conference on Mobile Computing and Networking*, Mobicom '12, pages 137–148, New York, NY, USA, 2012. ACM.

[260] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *Security Privacy, IEEE*, 5(2):32–39, March 2007.

[261] Chiachih Wu, Yajin Zhou, Kunal Patel, Zhenkai Liang, and Xuxian Jiang. Airbag: Boosting smartphone resistance to malware infection. In *Proceedings of the 21th Annual Network and Distributed System Security Symposium (NDSS '14)*, February 2011.

[262] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69, Aug 2012.

[263] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. The impact of vendor customizations on android security. In *ACM SIGSAC Conference on Computer & Communications Security*, CCS'13, pages 623–634, New York, NY, USA, 2013. ACM.

[264] Xueping Wu, Dafang Zhang, Xin Su, and WenWei Li. Detect repackaged android application based on http traffic similarity. *Security and Communication Networks*, pages n/a–n/a, 2015.

[265] Tobias Wüchner, Martín Ochoa, and Alexander Pretschner. Malware detection with quantitative data flow graphs. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 271–282, New York, NY, USA, 2014. ACM.

[266] XDA. Pingpongroot, May 2015.

[267] Mingyuan Xia, Lu Gong, Yuanhao Lyu, Zhengwei Qi, and Xue Liu. Effective real-time android application auditing. In *Security and Privacy (SP)*, pages 899–914, May 2015.

[268] Liang Xie, Xinwen Zhang, Jean-Pierre Seifert, and Sencun Zhu. pbmds: A behavior-based malware detection system for cellphone devices. In *Proceedings of the Third ACM Conference on Wireless Network Security*, WiSec'10, pages 37–48, New York, NY, USA, 2010. ACM.

[269] Luyi Xing, Xiaorui Pan, Rui Wang, Kan Yuan, and XiaoFeng Wang. Upgrading your android, elevating my malware: Privilege escalation through mobile os updating. In *Proceedings of the 35^{th} IEEE Symposium on Security and Privacy (S&P)*, SAN JOSE, CA, USA, May 2014.

[270] xray. Android vulnerabilities, 2014.

[271] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical policy enforcement for android applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 27–27, Berkeley, CA, USA, 2012. USENIX Association.

[272] Zhi Xu, Kun Bai, and Sencun Zhu. Taplogger: Inferring user inputs on smartphone touchscreens using on-board motion sensors. In *Proceedings of the Fifth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WISEC '12, pages 113–124, New York, NY, USA, 2012. ACM.

[273] Zhi Xu and Sencun Zhu. Semadroid: A privacy-aware sensor management framework for smartphones. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, CODASPY '15, pages 61–72, New York, NY, USA, 2015. ACM.

[274] 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 Conference on Security Symposium*, Security'12, pages 29–29, Berkeley, CA, USA, 2012. USENIX Association.

[275] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *19th European Symposium on Research in Computer Security (ESORICS'14)*, Lecture Notes in Computer Science, Wroclaw, Poland, 2014. Springer Berlin Heidelberg.

[276] Zhemin Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *Proceedings of the 2012 Third World Congress on Software Engineering*, WCSE '12, pages 101–104, Washington, DC, USA, 2012. IEEE Computer Society.

[277] Yanfang Ye, Dingding Wang, Tao Li, Dongyi Ye, and Qingshan Jiang. An intelligent pe-malware detection system based on association mining. *Journal in Computer Virology*, 4(4):323–334, 2008.

[278] Suleiman Y. Yerima, Sakir Sezer, Gavin McWilliams, and Igor Muttik. A new android malware detection approach using bayesian classification. In *Proceedings of the 2013 IEEE*

27th International Conference on Advanced Information Networking and Applications, AINA '13, pages 121–128, Washington, DC, USA, 2013. IEEE Computer Society.

[279] Apostolis Zarras, Antonis Papadogiannakis, Robert Gawlik, and Thorsten Holz. Automated generation of models for fast and precise detection of http-based malware. In *2014 Twelfth Annual International Conference on Privacy, Security and Trust (PST)*, pages 249–256, July 2014.

[280] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 1105–1116, New York, NY, USA, 2014. ACM.

[281] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: an efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 103–114, 1996.

[282] Xiao Zhang and Wenliang Du. Attacks on android clipboard. In Sven Dietrich, editor, *Detection of Intrusions and Malware, and Vulnerability Assessment*, volume 8550 of *Lecture Notes in Computer Science*, pages 72–91. Springer International Publishing, 2014.

[283] Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 611–622, New York, NY, USA, 2013. ACM.

[284] Yulong Zhang. Ssl vulnerabilities: Who listens when android applications talk?, August 2014.

[285] Shuang Zhao, Patrick P. C. Lee, John C. S. Lui, Xiaohong Guan, Xiaobo Ma, and Jing Tao. Cloud-based push-styled mobile botnets: A case study of exploiting the cloud to device messaging service. In *Comp. Sec. Applications Conf. (ACSAC)*, pages 119–128, 2012.

[286] Wu Zhou, Zhi Wang, Yajin Zhou, and Xuxian Jiang. Divilar: Diversifying intermediate language for anti-repackaging on android platform. In *Proceedings of the 4th ACM*

Conference on Data and Application Security and Privacy, CODASPY '14, pages 199–210, New York, NY, USA, 2014. ACM.

[287] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. Appink: Watermarking android apps for repackaging deterrence. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 1–12, New York, NY, USA, 2013. ACM.

[288] 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, New York, NY, USA, 2013. ACM.

[289] 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.

[290] Xiaoyong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. Identity, location, disease and more: Inferring your secrets from android public resources. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer &#38; Communications Security*, CCS '13, pages 1017–1028, New York, NY, USA, 2013. ACM.

[291] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and Xiaofeng Wang. The peril of fragmentation: Security hazards in android device driver customizations. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, SAN JOSE, CA, USA, May 2014.

[292] Yajin Zhou and Xuxian Jiang. Android malware genome project. http://www.malgenomeproject.org, 2012.

[293] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society.

[294] Yajin Zhou and Xuxian Jiang. Detecting passive content leaks and pollution in android applications. In *20th Annual Network and Distributed System Security Symposium, NDSS*, San Diego, California, USA, Feb 2013.

[295] Yajin Zhou, Kapil Singh, and Xuxian Jiang. Owner-centric protection of unstructured data on smartphones. In Thorsten Holz and Sotiris Ioannidis, editors, *Trust and Trustworthy Computing*, volume 8564 of *Lecture Notes in Computer Science*, pages 55–73. Springer International Publishing, 2014.

[296] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, February 2012.

[297] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming information-stealing smartphone applications (on android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST'11, pages 93–107, Berlin, Heidelberg, 2011. Springer-Verlag.

Publications Related to the Thesis

My contributions to the following papers are either partial or complete. I don't use percentage to avoid any misunderstanding for the rest of co-authors as a paper is not just the sum of contributions.

Minor Indication in this thesis

1. DLLMiner: Structural Mining for Malware Detection.
   Masoud Narouei, **Mansour Ahmadi**, Giorgio Giacinto, Hassan Takabi, Ashkan Sami
   April 2015, Security and Communication Networks, Wiley (See Section 1.3)
   My Contribution: Idea Proposal, Data Gathering, Doing Experiments, Writing

2. Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification
   Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov and Mikhail Trofimov, Giorgio Giacinto
   March 2016, 6th ACM Conference on Data and Applications Security and Privacy (CODASPY), New Orleans, USA (See Section 1.3)
   My Contribution: Idea Proposal, Data Gathering, Implementation, Doing Experiments, Writing

3. DROIDSCRIBE: Classifying Android Malware based on Runtime Behavior
   Santanu Kumar Dash, Guillermo Suarez-Tangil, Salahuddin Khan, Kimberly Tam, **Mansour Ahmadi**, Johannes Kinder and Lorenzo Cavallaro
   May 2016, Mobile Security Technologies (MoST), in conjunction with the IEEE S&P, San Jose, CA, USA (See Section 5.5)
   My Contribution: Implementation, Doing Experiments

The most relevant ones to this thesis

4. Clustering Android Malware Families by Http Traffic.
   Marco Aresu, Davide Ariu, **Mansour Ahmadi**, Davide Maiorca, Giorgio Giacinto
   October 2015, 10th IEEE International Conference on Malicious and Unwanted Software (MALCON) Puerto Rico, USA (See Chapter 3)
   My Contribution: Data Gathering, Writing

5. Detecting Misuse of Google Cloud Messaging in Android Badware
   Mansour Ahmadi, Battista Biggio, Steven Arzt, Davide Ariu, Giorgio Giacinto
   October 2016, Security and Privacy in Smartphones and Mobile Devices (SPSM), in conjunction with the ACM CCS, Vienna, Austria (See Chapter 4)
   My Contribution: Idea Proposal, Data Gathering, Implementation, Doing Experiments, Writing

6. DroidSieve: Fast and Accurate Classification of Obfuscated Android Malware
   Guillermo Suarez-Tangil, Santanu Kumar Dash, **Mansour Ahmadi**, Johannes Kinder, Giorgio Giacinto, Lorenzo Cavallaro
   Submitted to CODASPY'17 (See Chapter 5)
   My Contribution: Idea Proposal, Data Gathering, Implementation, Doing Experiments, Writing