Dynamic monitoring of Android malware behavior: A DNS-based approach

Oscar Somarriba Jarquín

Advisors:
Dr. Urko Zurutuza Ortega
Dr. Roberto Uribeetxeberria Ezpeleta
Department of Electronics and Computer Science

Mondragon University

A thesis submitted for the degree of

Doctor of Philosophy

May 2019

Acknowledgements

I am grateful to the many people whose support was indispensable to this work. Foremost I would like to express my deep gratitude to Dr. Urko Zurutuza Ortega, for giving me the opportunity to join the Department of Electronics and Computer Science group at Mondragon University (MU) and whose wisdom, guidance, patience, and encouragement have helped me achieve the seemingly impossible. I would like also to express my appreciation to my co-advisor, Dr. Roberto Uribeetxeberria Ezpeleta from MU as well; his enthusiasm, feedback and practical way of thinking has broadened my perspectives.

The financial support from the Erasmus Mundus Lamenitec Programme and the National University of Engineering (UNI) is gratefully acknowledged.

Special thanks to Dr. Iñaki Garitano and Iñaki Arrenaza for their valuable reviews and discussions. Also thanks to Dr. Enaitz Ezpelta and Dr. Mikel Iturbe for providing very important support during my wonderful stay in MU. Also I would like to thank to Assier Aduriz, Dr. Pablo Vásquez, Dr. Alain Peréz, Dra. Ane Alberdi, Dr. Aitor Lizeaga, Dr. Aitor Arrieta, Dra. Maite B., Dr. Raúl Roldan, Dr. Unai Bernabe, Miriam Moreno, Mikel M., contributed to my work in several ways too long to be listed here.

I am particularly indebted to Edurne Agirre, and Oihane Lameirinhas for their assistance with all practical issues regarding my work.

Last but not least, I want to express my gratitude to my two sons Oscar Alejandro & Jan Sebastian, to mother Angelita, to my nephew Douglas Alexander, and to my brothers and sisters, all of whom have been supporting me all this time and an example to follow.

Abstract

The increasing technological revolution of the mobile smart devices fosters their wide use. Since mobile users rely on unofficial or third-party repositories in order to freely install paid applications, lots of security and privacy issues are generated. Thus, at the same time that Android phones become very popular and growing rapidly their market share, so it is the number of malicious applications targeting them. Yet, current mobile malware detection and analysis technologies are very limited and ineffective. Due to the particular traits of mobile devices such as the power consumption constraints that make unaffordable to run traditional PC detection engines on the device; therefore mobile security faces new challenges, especially on dynamic runtime malware detection. This approach is import because many instructions or infections could happen after an application is installed or executed. On the one hand, recent studies have shown that the network-based analysis, where applications could be also analyzed by observing the network traffic they generate, enabling us to detect malicious activities occurring on the smart device. On the other hand, the aggressors rely on DNS to provide adjustable and resilient communication between compromised client machines and malicious infrastructure. So, having rich DNS traffic information is very important to identify malevolent behavior, then using DNS for malware detection is a logical step in the dynamic analysis because malicious URLs are common and the present danger for cybersecurity. Therefore, the main goal of this thesis is to combine and correlate two approaches: top-down detection by identifying malware domains using DNS traces at the network level, and bottom-up detection at the device level using the dynamic analysis in order to capture the URLs requested on a number of applications to pinpoint the malware. For malware detection and visualization, we propose a system which is based on dynamic analysis of API calls. This

can help Android malware analysts in visually inspecting what the application under study does, easily identifying such malicious functions. Moreover, we have also developed a framework that automates the dynamic DNS analysis of Android malware where the captured URLs at the smartphone under scrutiny are sent to a remote server where they are: collected, identified within the DNS server records, mapped the extracted DNS records into this server in order to classify them either as benign or malicious domain. The classification is done through the usage of machine learning. Besides, the malicious URLs found are used in order to track and pinpoint other infected smart devices, not currently under monitoring.

Contents

Chapter 1

Introduction

This chapter contains the antecedents and an overview of the research project that we carried out in order to develop this PhD thesis. First of all, we motivate the main reasons why we chose as subject of the thesis, **Dynamic DNS Monitoring of Android Malware behavior** as the main scope of this work. Second, we introduce and explain the background of the research, focusing briefly on the major issues of its knowledge domain and clarifying why these issues are worthy of attention. Third, we then proceed with the presentation of the research statement. Forth, afterwards, we link the research problem statement with the objectives of this work and with the hypotheses that guide us to provide a solution to the problem; as well as the research methodology used in this work is described. Finally, the thesis contributions with its associated publications are detailed, and the thesis outline is also presented.

1.1 Motivation

Mobile devices have become an attractive and an indispensable asset today, they are based on outstanding advances on computing & communication and sensing capacities, making critical personal and professional information accessible to the user at all times. As the most popular personal smart devices at present, smartphones are outselling the number of Personal Computers (PCs) worldwide since 2011 [2]. In our research we consider Android Operating System (OS), because is the dominant mobile operating system, nowadays. Given the ubiquitous nature of Android OS and the menaces against this mobile ecosystem, there is an urgent need for comprehensive & effective techniques to support the development of trustworthy tools for detection approaches and classification analysis. Since Android users are usually related with third-party applications, lots of security and privacy problems are generated. Yet, current mobile malware detection techniques and analysis technologies are still ineffective and limited. In [3], it is stated that “Due to the specific characteristics of mobile devices such as limited resources, constant network connectivity, user activities and location sensing, and local communication capability, mobile malware detection faces new challenges, especially on dynamic runtime malware detection. Many intrusions or attacks could happen after a mobile App is installed or executed.” However, both in academia and industry there is a pressing need for practical and effective dynamic malware detection approaches.

During the last decade, we have witnessed the rise of a new generation of personal devices that have revolutionized our modern society. The massive adoption of mobile communications in everyday life has brought unprecedented need for the society to trust in mobile infrastructures. This is a major challenge in terms of security today, since the amount of smart devices is increasing growth in the mobile market of smartphones and, despite the existing security mechanisms. In Figure 1.1, the growing and forecast of the mobile phones worldwide, 2015-2020. Concurrent with the aforementioned emergent growth is the amount mature malicious applications (commonly referred to as malware) targeting the smart devices.

Nowadays, the Android application ecosystem has grown considerably over the last years. The increasing number of malicious applications targeting Android devices raises the demand for analyzing them to find where the malcode is triggered when user interacts with them. In addition, smartphones running on Android platform represent an overwhelming majority of smart devices. For instance, with a global market share of 85% in the first quarter of 2017 (1Q17) and keeps growing continuously [4]. Besides, it was most placed at 3.5 million the number of available applications for mobile Android OS in the Google Play Store in December 2017 [5], see Figure 1.2. In fact, since every smartphone is actually a hand-held computer, it can be infected by malware. Of course, the outgrowth of mobile smart devices has also been supported by the enhancement of the OS technology upholding them. Therefore, in the Android ecosystem, the number of malicious Apps are constantly increasing. For instance, in [6], it is reported that a new malware for Android is released roughly every 10 seconds.

The F-Secure-Threat-Report State of Cyber Security 2017 [7] stated that "There are over 19 million malware programs developed especially for Android, making Google’s mobile operating system the main target for mobile malware. The reason

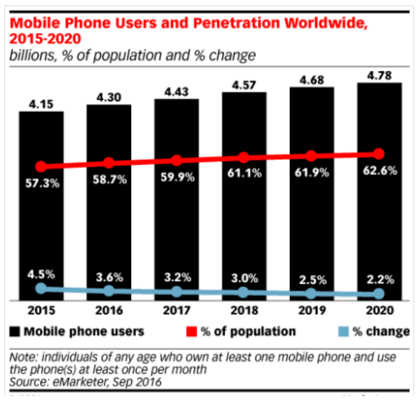


Figure 1.1: Mobile Phone Users and Penetration Worldwide, 2015-2020 [8].

for this is the vast distribution of Android devices, as well as the relatively open system for the distribution of Apps. And consequently, over 99% of all malware programs that target mobile devices are designed for Android devices." Figure 1.3 shows the reported number of Android Malware in AV-TEST's Database [7] from January 2011 up to part of April 2017. Also, another Threat Report [9] shows, see Figure 1.4, that between January and September 2017, it was found 32 different menaces on Google Play; around twice the amount from the same period in 2016. This seems like a trivial problem and it does not sound like a big number of devices, but this kind of threats are very smart and specialized malware accordingly to [9]: "the significant point was that this looked like targeted, precision malware rather than a broad data-stealing tool."

The increase in the number of Internet-connected mobile devices worldwide because of the portability and relatively low cost of the smartphones, along with a gradual adoption of LTE/4G, has drawn the attention of attackers seeking to exploit vulnerabilities and mobile infrastructures. Therefore, at the same time, the malware targeting smartphones has grown exponentially due the popularity of Android OS, which has led to a huge increase in the spreading of this kind of malicious applications. Moreover, mobile users increasingly rely on unofficial repositories to freely install paid applications whose protection measures are at least dubious or unknown to say to speak. Some of these applications have been

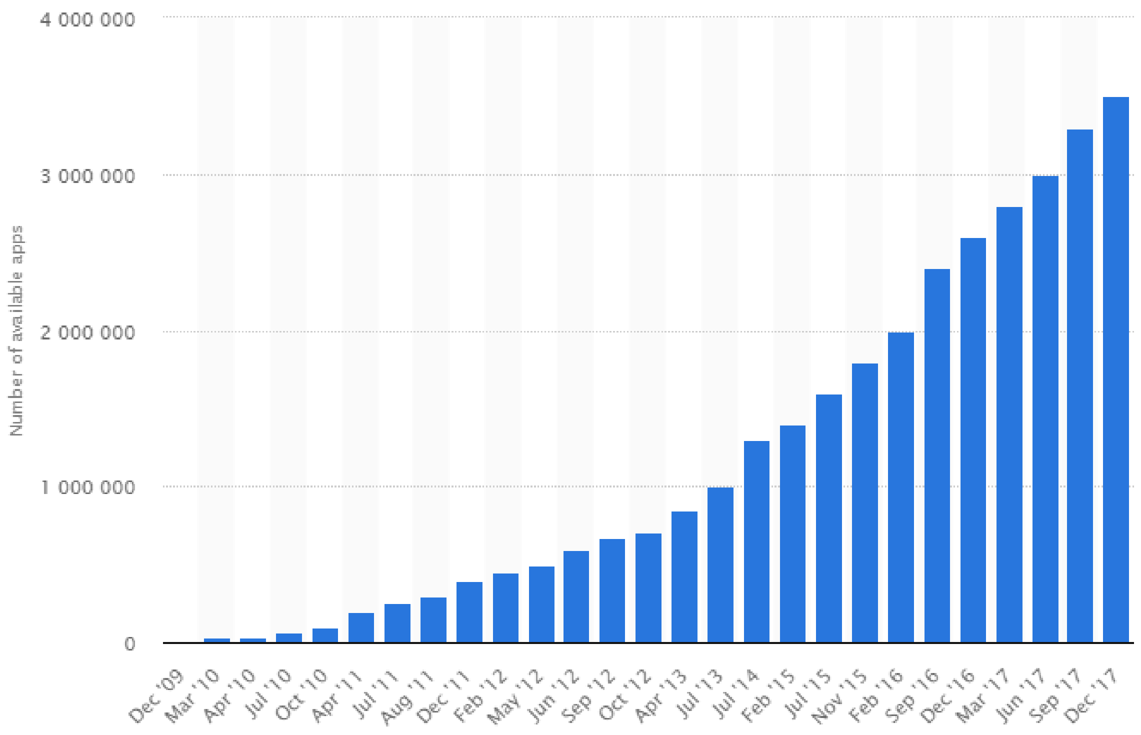


Figure 1.2: Number of available applications in the Google Play Store from December 2009 to December 2017 [7].

uploaded to such repositories by malevolent communities that incorporate malicious code into them. In agreement with [10]: "The end-users without enough knowledge on the security aspect of mobile applications cannot identify whether the downloaded App is malicious. These unverified and unreliable mobile applications may lead to the risk of devices hacking." This poses strong security and privacy issues both to users and operators [11].

In order to cope with the malware threats, it is required collecting a large amount of data issued by applications for smartphones, which is essential for making statistics about the applications' usage or characterizing the applications. Characterizing applications might be useful for designing both an anomaly detector and a misuse detector, for instance. So, it is necessary to record device information most efficiently and effectively possible, to face this malware increase. For example, logging information from applications destined for smartphones is becoming vital for evaluating the security of an application. A log may characterize the application behavior, e.g. for designing an anomaly detector or for evaluating the energy footprint.

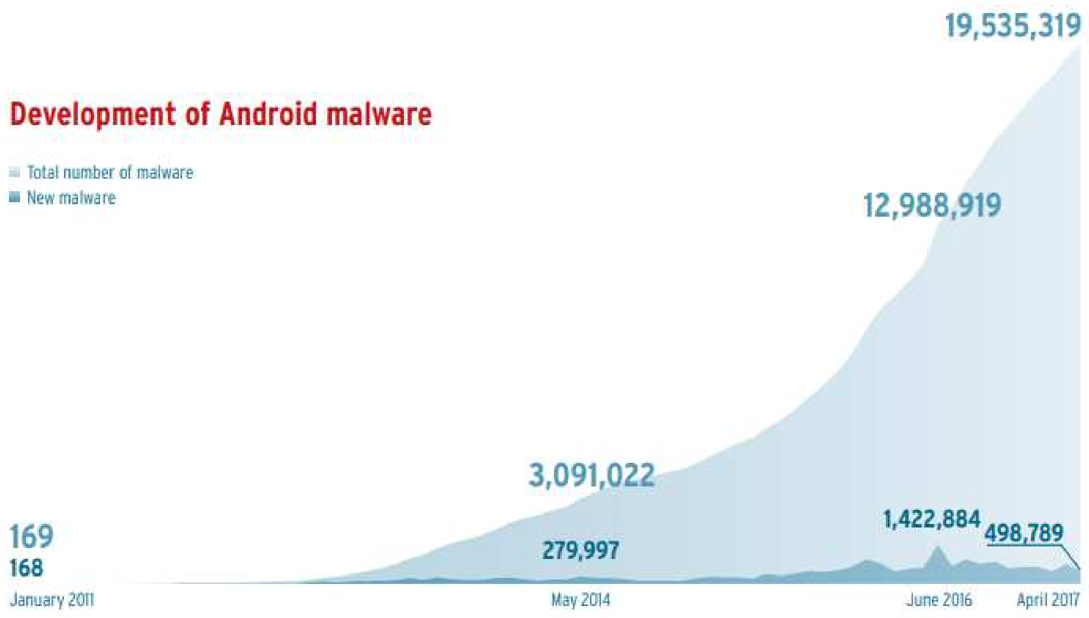


Figure 1.3: Number of malicious Android Apps reported to AV-TEST Institute, over the time, January 2011 to April 2017 [7].

However, Android platforms put restrictions on applications for security reasons. These restrictions prevent us from easily collecting traces without modifying the firmware or rooting the smartphone. Since modifying the firmware or rooting the smartphone may void the warranty of the smartphone, this method cannot be deployed on a large scale. Nevertheless, most of the methods presented in the literature so far need to root the phone in order to access relevant information. In this situation, many users may resist rooting a phone and thereby losing the warranty that comes with it, but may accept a purposeful study by a trusted agent (e.g., a security vendor).

This research work proposes an infrastructure for inserting hooks and/or with a non-rooted basic sniffer (Android application), collecting the application traces using the hooks and a network traffic agent, and uploading the traces to a remote server for observation and analysis. In this way, we are able to monitor Android applications (Apps) at a large scale at the application layer independently from the hardware, and without requiring changes to the firmware or rooting the phone. We will describe the infrastructure all the way from the client end elements to the elements that reside at the remote server in the coming chapters. Furthermore, we show how our potential changes at the application layer to achieve the mentioned remote monitoring may affect the application performance, by measuring its overhead at the smartphone side. Our monitoring system is portable between smartphones running on Android platforms. As opposed to other works [12], [13], we do not root the smartphone or change its firmware in order to monitor smartphones. The monitoring infrastructure can be made freely available to the Android security community, or adapted for other purposes.

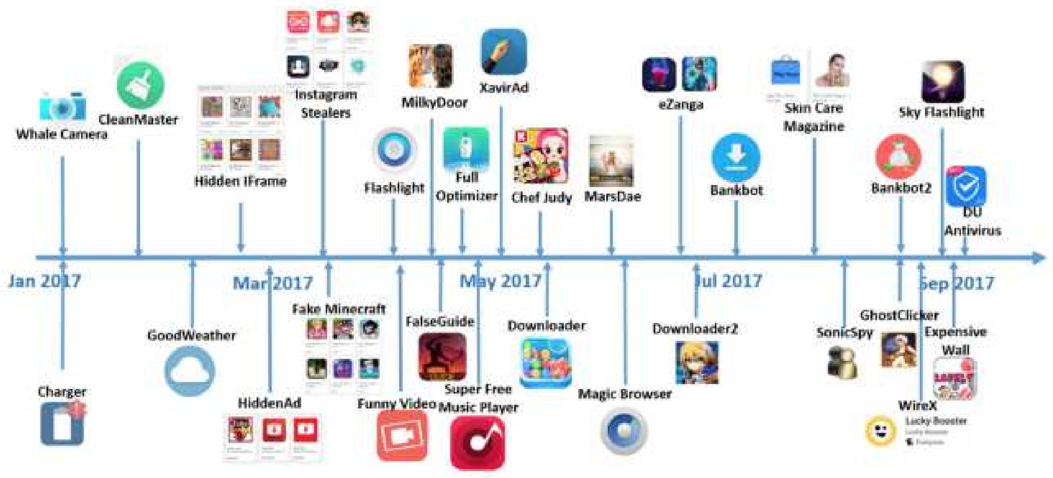


Figure 1.4: Threats on Google Play doubled between January and September2017 [9].

So, there is an open issue where research is needed: To monitor the current fast and ever-growing malware threat (problem), which is raising and ever increasing in number and complexity as the time goes on with more sophistication to evade detection and analysis.

On the other hand, in [14] it is mentioned that "greater and greater amounts of manual effort are required to analyze the increasing number of new malware samples. This has led to a strong interest in developing methods to automate the malware analysis process". Therefore, there is a need to continuously monitor applications and learn about their behavior on the "flight".

To our knowledge, many mobile users do not keep safe or protect their smartphones from malware since they falsely assume that the mobile network operator will safeguard them in the case of an attack. "However, this is not (yet) true. So far, only Internet Services Providers (ISPs) provide real-time malware detection directly in the network and mobile end users are left unprotected." [15]. Indeed, we would like to contribute in this direction, to promote the protection of smart devices from malware, in order to reduce this gap in the Android ecosystem security.

In summary, mobile devices have become major targets for smart malware due to their constant network activity, including the Internet access. Thus, there is an urgent need for detecting potential malicious behaviors by means of advanced dynamic mobile malware detection methods. Furthermore, Android malware is one of the major security issues and fast growing threats facing the Internet in the mobile arena. In this context, it is worth mentioning that the Domain Name System (DNS) is widely misused by miscreants in order to provide Internet connection within malicious networks, and DNS has turned an appealing target for malware developers. Thereby, DNS should be the first line of defense against many malevolent attacks. As it is well-known, DNS is one of the key critical elements of the Internet that facilitates associating or translating memorized domain names into IP addresses, and vice versa. Then subsequently, in addition to the crucial role in functioning of the Internet, DNS is put to wrong use by malware authors. Thus, the aggressors rely on DNS to provide adjustable and resilient communication between compromised client machines and malicious infrastructure [16].

Besides, the information collected from the devices along with the collection of DNS-service traffic in the networks of the operator might be combined for the monitoring, detection, characterization, and mitigation of mobile threats, as well as to create an early warning system for the operators. From an infrastructure perspective, the deployment of mobile devices is a double-edged sword: it can be used for massive attacks and rapid exploitation of security threats, but it can also be used as a massive network of distributed sensors in order to obtain a global real-time location for the emergence of malware and to facilitate early warning to the operator's infrastructure. This thesis will take advantage of the potential of this latest idea. In this way, it is noting that a mobile operator could quickly or indirectly detect other infected devices that had not installed the monitoring application. This detection is due to they behave in the same manner, by doing the same DNS queries than the monitored devices. This is certainly a very valuable benefit, because we do not need to monitor all the smart devices in the mobile network at the same time. Since we collect the used URLs on the Android device instead of on a remote server or gateway, we shorten the time to detect malware as it is suggested in the hybrid analysis method dubbed NeseDroid [17].

Consequently, it seems natural to link both issues, the dynamic analysis of the growing theats posed by Android malware that abused intensively of the DNS queries at the smartphone level with the DNS service traffic at the network level. Thus, this thesis aims to put together these two approaches by applying Dynamic DNS Analysis for Android malware detection.

1.2 Technical Background

In this section, we introduce what the reader needs to know to understand the context of the thesis.

1.2.1 Android Systems

In this subsection we introduce the Android OS. Android is a Linux-based OS designed primarily for touchscreen mobile devices developed by Android, Inc., which was later bought by Google.

In order to understand the Android malware features it is firstly important to have a whole picture of the security model of this mobile OS to be able to assess its strengths and vulnerabilities, and therefore have an idea of the deficiencies or aspects that need improving that are currently being exploited by malware writers. The architecture of Android is based on a multilayer model as shown in Figure 1.5.

Platform Architecture. The architecture of the Android System can be divided into four main parts (see Figure 1.5): applications, application framework, middleware, and Linux kernel.

- **Applications**: The top layer of the architecture is where the applications are located. They are written in Java language and use the APIs (Application Programming Interfaces) and libraries provided by the lower layers. An Android application is composed of several components, amongst which we have Activities and Services. Activities provide an user interface (UI) of the application and are executed one at a time; Services are used for background processing such as communication, for instance.

- **Application Framework**: This is a suite of Services that provides the environment in which Android applications run and are managed. These programs provide higher-level Services to applications in the form of Java classes.

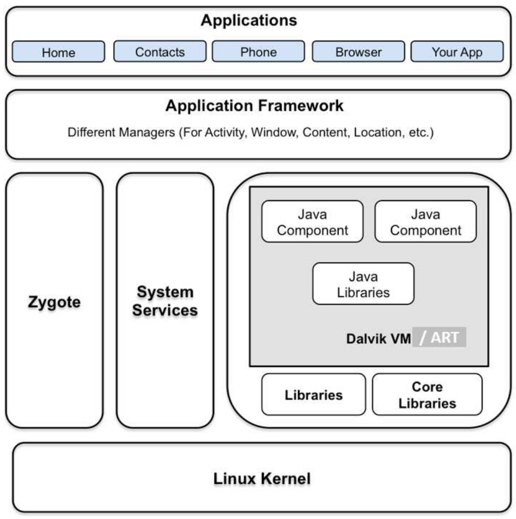


Figure 1.5: Overview of the Android System.

- **Middleware**: This layer is composed of the Android runtime (RT) and C/C++ libraries. The Android RT is, at the same time, composed of the Dalvik Virtual Machine (DVM) and a set of native (core) Android functions. Note that Android version 4.4 launches a new virtual machine called Android runtime (ART). ART has more advanced performance than DVM, among other things, by means of a number of new features such as the ahead-of-time (OTA) compilation, enhanced garbage collection, improved application debugging, and more accurate high-level profiling of the Apps [18]. The DVM/ART is a key part of Android as it is the software where all applications run on Android devices. Each application that is executed on Android runs on a separate Linux process with an individual instance of the DVM/ART, meaning that multiple instances of the DVM/ART exist at the same time. This is managed by the Zygote process, which generates a fork of the parent DVM/ART instance with the core libraries whenever it receives a request from the runtime process.

- **Linux Kernel**: The bottom layer of the architecture is where the Linux kernel is located. This provides basic system functionality like process and memory management. The kernel also handles the drives for interfacing to peripheral hardware such as screen and camera.

In standard Java environments, Java source code is compiled into Java bytecode, which is stored within .class format files. These files are later read by the Java Virtual Machine (JVM) at runtime. On Android, on the other hand, Java source code that has been compiled into .class files is converted to .dex files, frequently called Dalvik Executable, by the “dx” tool. In brief, the .dex file stores the Dalvik bytecode to be executed on the DVM.

Android applications are presented on an Android application package file (APK) .apk, the container of the application binary that contains the compiled .dex files and the resource files of the App. In this way, every Android application is packed using zip algorithm. An unpacked App has the following structure (several files and folders)[7], see Figure 1.6:

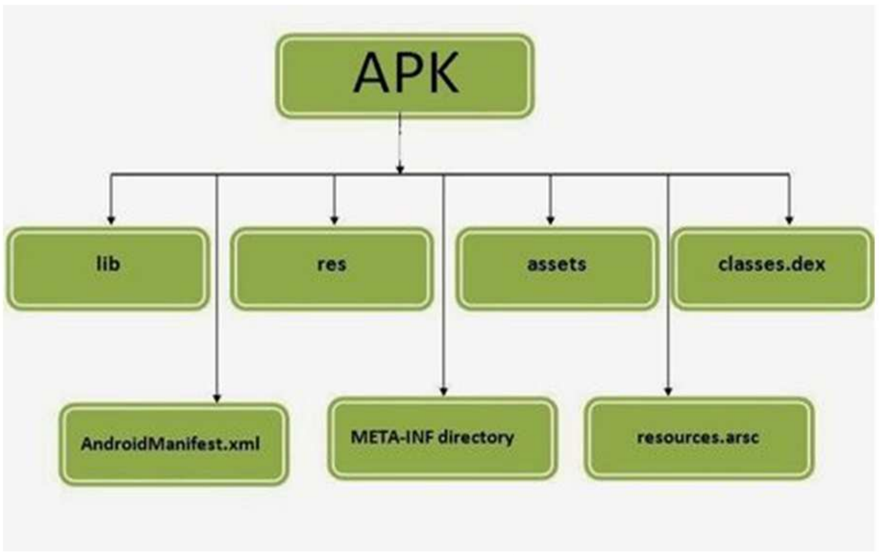


Figure 1.6: The APK package structure.

- **META-INF:** it is a directory which contains the digital certificate with which the application was signed (App RSA), and the signatures of all the files within the APK package (the MANIFEST.MF and the CERT.SF files contain the list of files and the SHA (Secure Hash Algorithm), in particular the SHA-1 hashes, of the files and of their declarations in the Android-Manifest.xml file, respectively). It also holds a list resources;

- **an AndroidManifest file.xml:** it contains the settings of the application (meta-data) such as the permissions required to run the application, version, referenced libraries, the name of the application, and the definition of one or more components such as Activities, Services, and Broadcasting Receivers, or Content Providers. Upon installing, this file is read by the PackageManager, which takes care of setting up and deploying the application on the Android platform;

- **an assets folder:** it stores noncompiled resources. This is a folder containing applications assets, which can be retrieved by AssetManager;
- **The classes.dex:** it stores all Android classes compiled in the dex file format to be executed on the DVM. In the case of ART, Dalvik bytecode is stored in an .odex file (pre-processed of .dex) [19];
- **a res folder:** it holds the resources used by the applications. By resources, we mean the App icon, its strings available in several languages, images, UI layouts, menus, and so forth. In short, it contains resources not compiled into resources.arsc [19];
- **resources.arsc:** it is a file that describes the precompiled resources (e.g. binary xml);
- **lib:** it is a directory which contains the compiled code that is specific to a software layer of a processor;

For protecting a smartphone against attacks, several security mechanisms can be found into the corresponding OS. In particular, one the prevalent security mechanism in Android smartphones, is its used User's Permission model. Regarding the runtime permissions, note that the release of Android version 6.0 (API level 23), demands asking users for dangerous permissions during runtime or at the time the users are needed. Another security mechanism is the App execution isolation using the DVM/ART. In other words, Android uses UNIX-like user IDs to assign specific permission to applications. A comprehensive guide to Android security mechanisms can be found in [20] and [21].

1.2.2 Taxonomy of Mobile Malware

In this subsection, we introduce key aspects of mobile malware and list some of them. In our case, we will describe various potential attack scenarios where an attacker can take advantage of the vulnerabilities of the Android platform to compromise a user.

At present, the cyber attacks can be of any form ranging from opportunistic attacks such as phishing, spamming, SQL Injection Attack, Denial of Service (DoS), to more precisely targeted ones like the Advanced Persistent Threat (APT), which is an stealthy and continuous attempt or hacking process to infiltrate a particular target organization for business or political reasons.

In general, we can usually consider three common types of malicious or intrusive software (malware): virus, worm, and Trojan horses. A *virus* is a piece of code that can replicate itself. In other words, it has the ability to harm and self-replicating in order to infect hosts. A virus comes in a resident medium that can be, e.g. an executable file or a USB memory. If the user runs this "payload" (which represents the actual content that is used to harm the device of the victim), the virus executes its malicious commands, which can be almost everything the OS allows. A *worm* is a program that makes copies of itself and it can often spread without user interaction. Once started, it looks for an infectable victim within reach. If a victim is found, it tries to exploit a vulnerability to stick to the victim and then repeats this action. Sometimes worms allow back door access by dropping other malware to the infected machine. A back door allows a hacker to gain access to a remote computer. Malware can also come packaged as a *Trojan horse*, a software that appears to provide some useful functionalities but, it contains a malicious program instead. Moreover, a *Trojan horse* is a program that is used to trick users, e.g. as a popular application, in order to convince a user to execute or install it [22, 23].

The malware propagation concept refers to the electronic method, by which, malware is spread to an information system, platform or device it seeks to infect. Malware can be propagated using several techniques and communication interfaces (through OS, across wireless networks, by means of file sharing, through visualized systems, over e-mail communications, throughout social networking), in other words, ranging from an exploit to using social engineering (malware requiring user interaction) [23]. With respect to smartphones, most used means of infection are Bluetooth, Internet, SMS, MMS, Memory Card, and USB [22].

Nowadays, a universally accepted mobile malware taxonomy does not exist either in the academic literature or in the security experts domain. We can start by taking into consideration the taxonomies implicitly proposed by the major mobile anti-virus companies in their periodical reports.

Classification of Android malware attacks and intrusions. Regarding security and privacy issues to users, smart devices present greater challenges than conventional PCs. Of course, most of the existing Android malware types are directly inherited from the desktop space (e.g. Adware). However, there are other Android malware types that are unique to mobile space (e.g. Trojan-SMS). As aforementioned, this is due to such devices incorporate several sensors that could leak highly sensitive information about their owners or users [11]. On August the 9th 2010, Kaspersky discovered the very first SMS Trojan for Android in the wild dubbed *FakePlayer* [1].

By the end of 2010, the Android malware dubbed *Geinimi* was discovered. The paper in [24] presents a comprehensive Android malware evolution from the first SMS Trojan discovered in the wild in 2010 to the sophisticated malwares seen in the official Google Play during the first half of 2011 like DroidDream, DroidKungFu, and Plankton. Also, Castillo in [24] presents some common methodologies and tools used to analyze two samples of Android malware, namely: FakePlayer and Plankton.

La Polla et al. [25] conducted a literature survey on menaces and exposures with a focus on work published from 2004 until 2011. Furthermore, other authors also surveyed on threats and vulnerabilities following the same line of research as in [25] for the period from 2010 until 2013 [11], and from 2010 until 2014 [26], respectively. Several works related to the categorization and classification of malware attacks and intrusions can be found in [11, 27, 28, 29], and they included various of the taxonomies focused on the categorization of mobile device misuse. A thorough survey of Android malware is presented by Jiang and Zhou in [30] charting the most common types of permission violations in a large data set of malware. Noting that the Android Malware Genome (MalGenome) Project created circa 2011 has been one of the most widely studied dataset by the research community due to its easy access. However, by end of 2015, the Genome authors have stopped the efforts of malware dataset sharing due to resource limitation [3]. A more in-depth analysis of a current Android malware dataset public available is provided in [31]. Recently, in [32, 33, 19], they surveyed mobile malware analysis techniques to cope with Android malware on mobile devices, comprehensive taxonomies to classify and characterize the state-of-the-art research in this area, as well as malware tactics to hinder analysis.

An appealing alternative of malware classification encompassing most of the above described vulnerabilities is shown in Figure 1.7. This Figure also includes some malware usually directly inherited from the traditional PCs environment

[1]The list of one year of Android malware after it began is detailed in http://hackmageddon.com/2011/08/11/one-year-of-android-malware-full-list/

(e.g., Rogue-also known as FraudTool-AV), even if some of them have additional capabilities due to the mobile space.

Furthermore, the basic level premium-SMS Trojans is still expected to grow in number. Most seriously, current Trojans are using advanced polymorphism and metamorphism based techniques making it impossible to detect them solely through static analysis (i.e., only by means of code checking of the involved App, see further details in subsection1.2.3). Android malware with kernel-level rootkit has been demonstrated as a proof-of-concept already. Such malicious Apps are harder to combat since they are able to modify OS level code of the system. Researchers in [34] predict worms capable of self-replicating without human intervention as the next step in the evolutionary development of malware.

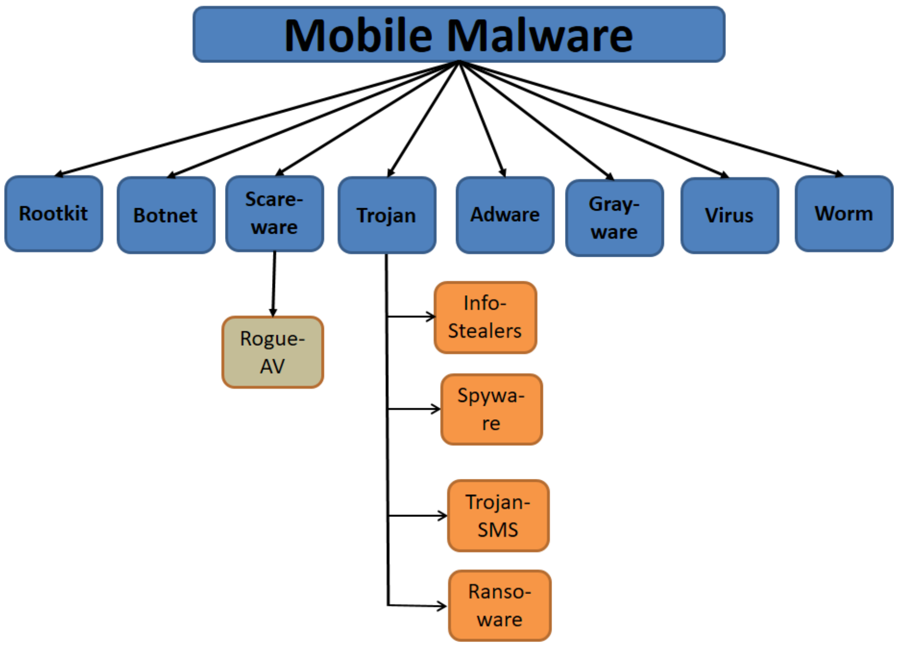


Figure 1.7: The Android malware types [35].

Some malware categories could be defined as follows [35], see Figure 1.7:

- *Root Exploits or Rootkits*, it is a malware that operates at the kernel level of the Android OS;
- *Botnet*, it is a number of interconnected infected smart devices. Each mobile device is called a bot. A collection of such bots is referred to as a botnet. They

execute tasks and clear their actions based on a control and command (C&C) server or software;

- *Rogue or Scareware*, it is a faked antivirus tool or a deceptive App that misleads a user to believe that it is a well-known or trusted software in order to steal money and/or confidential data;
- *Trojan or Trojan Horse*, it is a type deceptive App that is often disguised as legitimate software;
- *Infostealers*, Trojan that exploits Apps. Since, they can easily get the list of contacts of the user, browsing history, device International Mobile Equipment Identity (IMEI), etc. through API calls if they have the right permissions;
- *Spyware*, it is a Trojan that performs espionage on any actions of smart device users;
- *Ransonware*, it is a Trojan that usually prevents users from accessing their smartphone, either by locking or encrypting the users' files unless a rescue is paid;
- *Trojan-SMS*, Trojan that subscribes the users to premium-rate call services and the cost of these SMS messages are charged to the sender's phone bill, without his/her authorization.
- *Adware*, i.e., a program that displays unwanted advertisement [36];
- *Grayware*, which is a legitimate App that collects data user for the purpose of marketing or user profiling without harming intentions.
- *Virus/Worm*, a virus/worm adapted to the mobile environments.

In Figure 1.8 [37] shows the many different ways that a hacker can profit from a compromised mobile device. Some of these, such as ransomware, fake anti-virus sofware, botnet activity and data theft, have migrated from the traditional PCs.

Similar to other mobile platform owners do, such as iOS and Windows Phone, in Android the applications submitted to the official repository are analyzed before they are publicly available. In this particular case, Google runs those applications for a short period of time on a virtual machine called "Bouncer" (which is a dynamic analysis tool), looking for malicious behavior. However, usually the Bouncer

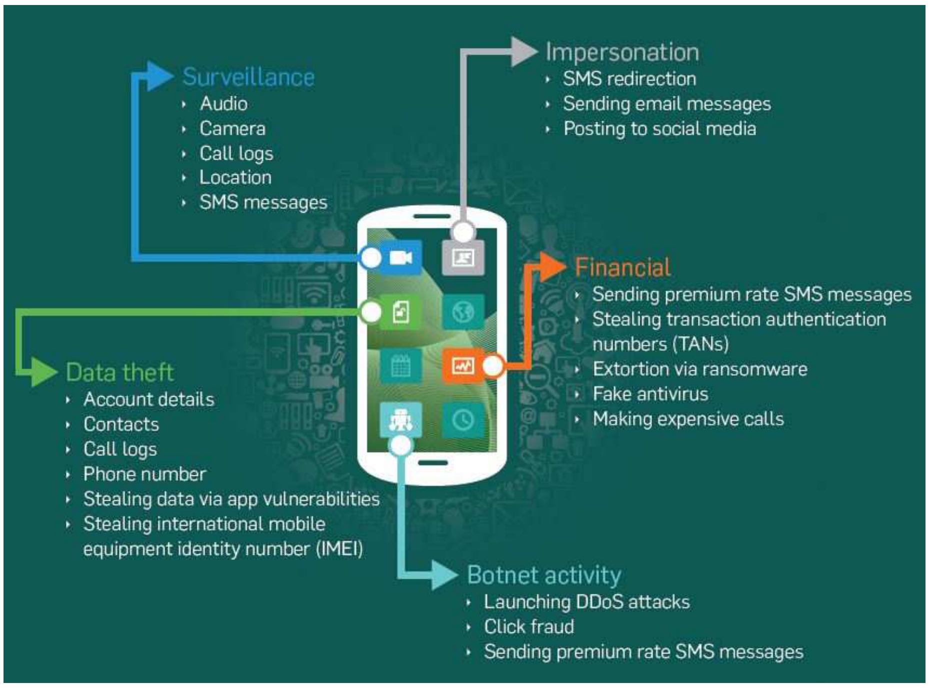


Figure 1.8: Anatomy of a hacked Mobile Device [37].

can be bypassed or tricked [11] since the malware could manage to circumvent the service by changing their manner of operating or with application updates and, apparently, all those mechanisms are not effective enough.

1.2.3 Malware analysis techniques

In [38] it is defined that "Malware analysis is the art of dissecting malware to understand how it works, how to identify it, and how to defeat or eliminate it."

Malware analysis is the study or process of extracting information from malware through code checking and/or code execution inspection by using different tools, techniques, and processes. It is a methodical approach to uncovering a malware's main purpose by extracting as much data from a given malware sample such as a virus, worm, botnet, scareware, spyware, trojan horse, rootkit, or backdoor. A method or a particular way of doing the malware analysis, that needs practical skills accordingly to a well established procedure is dubbed malware analysis technique. However, the malware analysis can be also considered an art or craft, because of the successful data extraction or to extract as much information from the malicious code, it depends on the capability/adaptive nature of malware sample under inspection and the experience/skills of malware analyst.

Most often, when performing malware analysis, we will have only the malware executable, which will not be human-readable. In order to make sense of it, we will use a variety of tools and tricks, each revealing a small amount of information. We will need to use a variety of tools in order to see the full picture. There are two fundamental approaches to malware analysis: static and dynamic. Static analysis involves examining the malware without running it. Dynamic analysis involves running the malware.

Static Analysis (SA) represents an approach of checking source code or compiled code of applications before it gets executed. In other words, this type of technique attempts to identify malicious code by unpacking and disassembler the malware. The results depend on the up-to-dateness of the corresponding detection rules and methods. Yet the static analysis can be evaded through obfuscation or encryption technique, and it is very ineffective against sophisticated malware. So, the main idea behind using static techniques for detecting possible malicious behavior is to utilize a relatively fast approach such as parametric-static code analysis, taint tracking, and control flow dependencies. Of course, all of these without actually executing the malware. However, SA can miss important behaviors of the malware.

Conversely, dynamic analysis techniques seek to identify malicious behaviors after deploying and executing the malware on a controlled device or on an emulator. Usually the dynamic analysis techniques focus on black-box testing. Blackbox testing is the process of executing a malware in order to monitor its behavior. Dynamic Analysis (DA) techniques considers parameters including network traffic, native code, system call sequences, processes, file system and registry changes, and user interaction. It usually involves running the malware in a isolated environment to track its execution behavior, therefore immune to obfuscation attempts. Egele [34] provides a comprehensive survey of various automated dynamic analysis techniques. While considered more effective against several polymorphic and metamorphic malwares which evade static analysis, dynamic analysis suffers from having very resource-intensive utilization.

In summary heretofore we have introduced, malware detection (MD) techniques for smart devices can be classified according to how the code is analyzed, namely: static analysis and dynamic analysis. In the former case, there is an attempt to identify malicious code by decompiling/disassembling the application and searching for suspicious strings or blocks of code; in the latter case the behavior of the application is analyzed using execution information. Examples of the two named categories are: *Dendroid* [39] as an example of an static MD for Android OS devices, and *Crowdroid* as a system that clusters system call frequency of applications to detect malware [40]. Also, hybrid approaches have been proposed in literature for detection and mitigation of Android malware. For example, Patel et al. [41] combine Android applications analysis and machine learning (ML) to classify the applications using static and dynamic analysis techniques. Genetic algorithm based ML technique is used to generate a rules-based model of the system.

1.2.4 Introduction to the Domain Name System

The Domain Name System (DNS), it is charged of handling all naming virtually within the Internet. It is a hierarchical decentralized system and core to the appropriate operation of hardly all Internet Protocol (IP) network applications, as well as the naming system for computers, smartphones, tablets, services, or other resources connected to the Internet. DNS is a networking system which is a fundamental element of the Internet functionally that provides the lookup service to convert domain names to their corresponding IP addresses.

The unavailability of the DNS network service due to a network disruption or product of a cyber-security or privacy attack, as well as a consequence of manipulating the integrity of the data contained within the DNS traffic, can collapse the network from the perspective of the end user, even though we have network connectivity (unless, of course, we already know the IP address of the web site we would like to connect straightforward to); however we will not be able to connect, and we will be unable to watch any hyper-linked contents. Thereby, one of the critical aspects to sustain the proper functioning of the whole Internet is the need to keep the DNS safe. Note that DNS query log files from the DNS servers provide clues of the security of this system [42].

Most of the current applications and malware are also using DNS tunneling techniques and HTTPs traffic. So, the DNS could be used as some kind of a backdoor. In this way, as several studies proposed, DNS is the first step in allowing users to connect or visiting to specific websites. Indeed, it is likewise usable by malware writers to carried out their malevolent activities. Given that the DNS traffic is always available to flow freely through networks, exposing networks to attacks that leverage this freedom of communications for lookups or for tunneling of data out of the mobile users [43].

Now, let us consider DNS network traffic. The domain name space is structured like a tree and it outlines the specifications for each of the nodes within the networked environment. A domain name identifies a node in the tree. The set of resource information associated with a particular name is composed of resource records (RRs), which is the elementary type of information link in DNS. Also, DNS defines a number of various types of RR. For instance, an A-type RR links a domain name with an IPv4 network address. The depth of a node in the tree is sometimes referred to as domain level. For instance, the domain name H.D.B.A. identifies the path from the root "." to a node H in the tree. Here, A. is a top-level domain (TLD), B.A. is a second-level domain (2LD), D.B.A. is a third-level domain (3LD), and so on [44] [45].

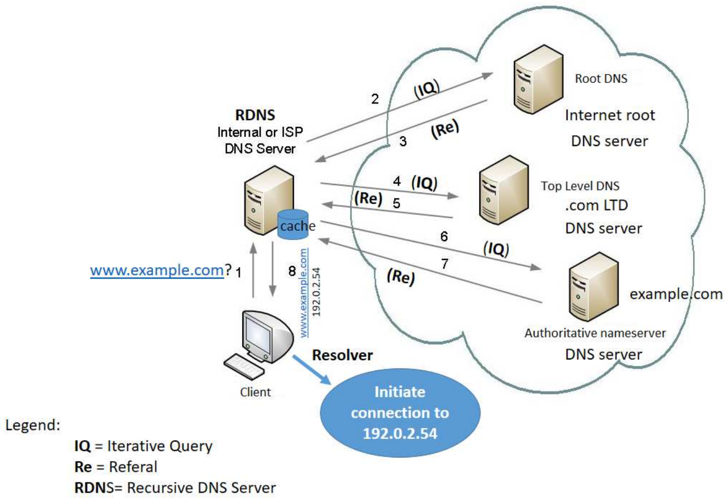

RDNS
Internal or ISP DNS Server
cache
www.example.com? 1
8 www.example.com 192.0.2.54
Client
Resolver
Initiate connection to 192.0.2.54
2 (IQ)
3 (Re)
4 (IQ)
(Re) 5
6 (IQ)
7 (Re)
Root DNS
Internet root DNS server
Top Level DNS .com LTD DNS server
example.com
Authoritative nameserver DNS server
Legend:
IQ = Iterative Query
Re = Referal
RDNS= Recursive DNS Server

Figure 1.9: Basic DNS Resolution Flow [44].

As stated in [45]: "RRs are returned in response to a DNS query from a requester. Figure 1.9 shows the basic flow of a DNS query. In this process, upon entry of the desired destination from a host for the A-type record for *www.example.com* in this case. A DNS query is initiated by a DNS resolver (usually included with the device OS) running on a host. This application is responsible for generating some sequence of queries and translating the responses to arrive at the requested resource. There are two parts in a typical DNS resolution request: the recursive and iterative part. In a typical use, an end system will issue a recursive request using a stub resolver to a dedicated recursive DNS resolver (RDNS) (Step 1, Figure 1.9). If not relevant information exists in the resolver cache the device will query the RDNS. So, in a recursive request, the RDNS is charged with completing the iterative portion of the DNS resolution process. It will communicate with the necessary remote name servers (NS) or DNS servers and returns a DNS answer, from the authoritative NS for the requested domain, to the stub resolver in the form of an RR-set. In the case of Fig.e 1.9, the RDNS sends iterative requests to the various levels of the DNS hierarchy (Steps 2 to 7). In Step 7, the RDNS receives the authoritative answer for www.example.com., and sends it to the requester (or stub resolver) in Step 8, completing the DNS resolution, and initiate connection to obtained IPv4, e.g. 192.0.2.54. The RDNS will typically cache the RR locally for up to some period, the Time To Live (TTL), specified in the RR."

Wessels et al. [46] were the first to analyze Domain Name System (DNS) query data as seen from the upper DNS hierarchy. The authors focused on examining the DNS caching behavior of recursive DNS servers from the point of view of AuthNS (Authoritative name servers) and TLD servers, and how different implementations of caching systems may affect the performance of the DNS. AuthNSs that have complete knowledge about a zone (i.e., they store the RRs for all the nodes related to the zone in question in its zone files) are said to have authority over that zone. Several studies provide deep understanding behind the properties of malware propagation and botnet's lifetime [44] [47]. An interesting observation among all these research efforts is the inherent diversity of the botnet's infected population. Besides, in [48], there is a proposal of the analysis of passive DNS traffic for network-based malware detection mainly for botnets, that make use of dynamic mapping (between domain names and IP addresses) known as domain-flux and IP-flux. Their analysis is based on graph theory and machine learning algorithms. Recently, a more in-depth DNS traffic analysis on these evasive techniques (agile DNS mappings) of the malware is given in [16]. Also, in [49] there is a report on DNS lookup patterns measured from the .com TLD servers. Their analysis shows that the resolution patterns for malicious domain names are sometimes different from those observed for legitimate domains. In [44], the system Kopis is proposed to monitor query streams at the upper DNS hierarchy and be able to detect previously unknown malware domains. Kopis directly uses the intuition behind these past research efforts in the requester diversity and requester profile statistical feature families in order to analyse DNS query patterns at the AuthNS and TLD server level for the purpose of detecting domain names related to malware. For instance, a low rate (less than 0.0009%) of malware infections of DNS traffic is reported by [50]. Regarding this result, in [19], it is stated that: "However, this method indirectly measured domain-name resolution traces." So, we need to further research in the Android ecosystem, the possible correlation between malware domain detection at the DNS-network service level and malware detection at the App level.

1.3 Research Statement, Hypotheses and Main Objective

In this section, we introduce the Problem Statement, the hypotheses that guide us to tackle the research work, and the principal objective of this thesis.

1.3.1 Research Statement

This thesis aims to deal with the current and emerging threat of the Android malware in mobile ecosystems. Much of the research reported in the literature surrounding mobile malware has been centered around the in-depth analysis of malicious Apps (host-based systems) rather the network-based [51]. However, smart devices are designed to be connected to a mobile network most of the time, so with the former approach we are not able to detect malware activities occurring on mobile devices through the Internet. Besides, with the latter approach (network-based systems) for instance, where applications could be also analyzed by observing the network traffic they generate, which enables us to detect malicious activities occurring on the smart device (e.g., the DNS queries from the smartphone to malicious remote server). Further by inspecting whom, an Android application connects to, we can deduce its malicious behavior [13]. Since most of the Android malicious Apps communicate with some command and control (C&C) servers using the DNS system, then the DNS analysis is an appealing way to detect the presence of malware.

As stated before, the cyber criminals make use of DNS services to exploit their malicious networks (e.g., botnet). Thus, at this network level analyzing DNS requests may be useful in the identification of current and future mobile threats. Furthermore, mobile devices have become major targets for malware due to their constantly crossing physical and network domains, so they are exposed to more rate infections than desktop environment. In addition to, DNS plays a crucial role in a large number of apps including those that allow the communication of most of the mobile malware with some remote server(s). Then, gathering DNS-service network traffic adds a new dimension to support the mobile malware detection. Developing a collaborative framework between the aforementioned approaches (top-down detection by identifying malware domains using DNS traffic and bottom-up detection using dynamic analysis on Apps to pinpoint the malware) could further enhance our chances of malware detection.

Furthermore, Android platforms put limitations on Apps for security reasons, in order to avoid easily gathering traces without reshaping the firmware or rooting the smartphone. Even so, we need to collect traces from Apps on a large scale, herein this will be done under two constraints such as preclude modifying the firmware or rooting the smartphone. As said before, without these restrictions the warranty of the smart device may be invalidated or further exposed to malware attacks. In particular, our research work faces the lack of a light framework for analyzing mobile malware and correlating these two levels, namely: application traces at host and network level and the traces of the DNS-service network traffic at the operator's infrastructure.

1.3.2 Hypotheses and Main Objective

The main goal of this thesis is to explore, design, and develop techniques that can be used to detect malicious mobile behavior from "massive" sets of heterogeneous sources. In particular, the DNS traffic activity produced by mobile malware will be inspected and correlated with device-related activity.

The following hypotheses of work will be considered for this research project:

- **Hypothesis 1**: Detection and visualization of Android Malware Behavior can be achieved by means of a rule-based system based on API calls without compromising the performance of the mobile devices involved.

- **Hypothesis 2**: It is possible to design and develop a system capable of extracting from the Android application, the DNS request (queries) monitoring the Android malware, in an efficient way regarding the resources of the smart device.

- **Hypothesis 3**: The execution of malware implies in most of the times the access of the same malicious domains in Internet, reason why the capture of this type of consultations can give rise to detect infected devices without needing to monitor the own device.

Hypotheses 1, 2 and 3 will be addressed in the coming Chapters 2, 3, and 4, respectively.

1.4 Technical Objectives

The most technical objectives of this thesis are described below:

1. Investigate existing Android malware detection algorithms and methods in terms of computing cost and detection strategy.
2. Design and develop a system capable of extracting from the Android application, the DNS traffic in an efficient way regarding the resources of the smart device.
3. Identify within the DNS records those requests that utilize the device running the Android application under test.
4. Map the local (extracted) DNS records into a server in order to classify them either as benign or malicious domain.
5. Develop a framework that automates the former analysis (items 2, 3, and 4) using our client(s) App(s) for smart devices developed or enhanced in this thesis.

1.5 Methodology

This chapter contains the methodological part of the research project. In this section the most important research stages are described.

The details of the methodology are defined according to the characteristics of design science research (DSR) [52, 53, 54, 55].

In order to carry out this thesis work, we have followed the guidelines for performing DSR in information systems as described by Peffers et al. in [56], see Figure 1.10. The methodology consist of six process steps or activities described as follow. The Figure 1.10 summarizes the steps, methods and techniques used in this research in accordance to the stages suggested by Peffers et al.:

1. **Problem identification and motivation:** In this activity, the definition of the specific research problem was taken as input and used the information gathered from the literature survey to motivate the relevance of the study and

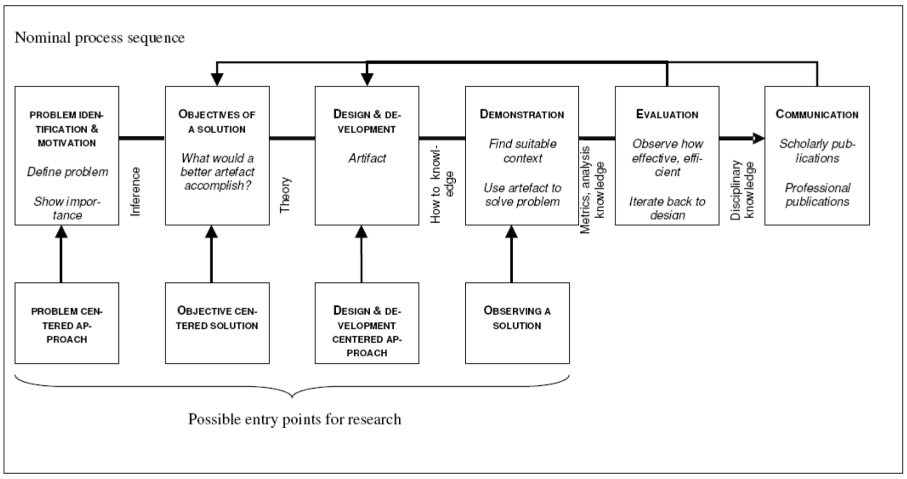

Nominal process sequence
PROBLEM IDENTIFICATION & MOTIVATION
Define problem
Show importance
Inference
OBJECTIVES OF A SOLUTION
What would a better artefact accomplish?
Theory
DESIGN & DEVELOPMENT
Artifact
How to knowledge
DEMONSTRATION
Find suitable context
Use artefact to solve problem
Metrics, analysis knowledge
EVALUATION
Observe how effective, efficient
Iterate back to design
Disciplinary knowledge
COMMUNICATION
Scholarly publications
Professional publications
PROBLEM CENTERED APPROACH
OBJECTIVE CENTERED SOLUTION
DESIGN & DEVELOPMENT CENTERED APPROACH
OBSERVING A SOLUTION
Possible entry points for research

Figure 1.10: Design science research process (DSRP) model, [56].

justification of the solution (e.g., it includes motivation and reasoning with the understanding of the problem by the researcher). The survey was also a tool to acquire useful knowledge to develop the proposed solution approach. The resources needed for this activity include knowledge of the state of the art problem and the importance of its solution. Here, we identified the problem to be resolved and we stated it clearly. See subsection 1.3.1.

2. **Define Objectives of a solution:** The second activity infers the outlined objectives of a solution from the problem definition and its requirements. Also, it includes a more comprehensive state-of-the-art report to extract potential open research issues, conceptual framework, and strategies to address the problem, and thus suggest a possible solution to the problem stated based on the formulated hypotheses. Herein, it is of paramount importance to take into account the current solutions and their efficacy, if any. See subsection 1.4.

3. **Design and development:** According to the authors in [54] the third activity consists: "Create the artifactual solution. Such artifacts are potentially, with each defined broadly, constructs, models, methods, or instantiations." For instance, we can establish the technique of implementing the artifact. Also in agreement with Peffers et al. in [56]:" This activity includes determining the artifact's desired functionality and its architecture and then creating the actual artifact. Resources required moving from objectives to design and

development include knowledge of theory that can be brought to bear as a solution." The design and prototyping of the artifact(s) usually implies some publications, see section 1.6.2.

In our case, we proposed a technique of systematically exploring and monitoring the execution of instrumented Apps in the cloud to detect Android malware. In addition to that, we designed and created an Android DNS sniffer to conduct network packet analysis. So, with the latter artifact, we can intercept malicious URLs based on the capture and review of the DNS queries done by the malware at the smart device, and correlated with DNS-service network traffic at the operator's infrastructure to find and locate other infected smart devices.

4. **Demonstration (Experimentation):** As stated in [56], this stage of *Design and Creation* process is about to: "Demonstrate the efficacy of the artifact to solve the problem. This could involve its use in experimentation, simulation, a case study, proof, or other appropriate activity. Resources required for the demonstration include effective knowledge of how to use the artifact to solve the problem." This activity is an iterative process which includes a solution to re-design or design and experimentation steps. Once the solution is envisioned, we plan to design and develop it. The outputs at this point are malware detector artifacts (i.e., to provide a dynamic analysis and tracking traces of audit data such as network traffic) and related publications, which validate the original design and development.

   We conduct the demonstration through experiments, so two frameworks were developed in this thesis. One of them, it is a framework named *AppShaper* (Chapter 2), which the main objective is the detection and visual analysis of Android malware. The first framework is a rule-based visualization of API calls of Android malware. The second framework dubbed *SIMPLEDNS* (Chapter 4) includes the DNS request monitoring of Android malware based on one of two novel methods proposed here (Chapter 3) combined with dynamic analysis of the Apps running on the smartphone [57, 58]. In the sequel, the analysis of the DNS-service network logs, including the identification of the infected devices in the operator network without the direct monitoring of them. The operation of *SIMPLEDNS* is described in Chapters 3 and 4 that follow. We also address the efficacy of the two frameworks developed to solve the problem, especially in the client side (i.e., the smart device). See Step 4, in Figure 1.11.

5. **Evaluation:** The evaluation activity validates the solution. In this stage, we employ a measurement campaign based on relevant metrics and analysis techniques and to characterize the performance of this approach (satisfying or not the stated hypothesis). In other words, we observe and measure how well the artifact supports a solution to the problem. Besides, if it is feasible we can conduct test on real scenarios, thereafter we can try to publish those results (if that is allowed). For further details, see step 5 of the Figure 1.11.

6. **Communication:** Finally, Peffers et al. [56] define this activity as: "Communicate the problem and its importance, the artifact, its utility and novelty, the rigor of its design, and its effectiveness to researchers and other relevant audiences, such as practitioners, when appropriate." Thus, we published the results of our research in a set of papers [35, 59, 60, 61, 57]. The final output of this activity is this thesis.

In Figure 1.11, we can appreciate in more details the research strategy that we utilize in this thesis work.

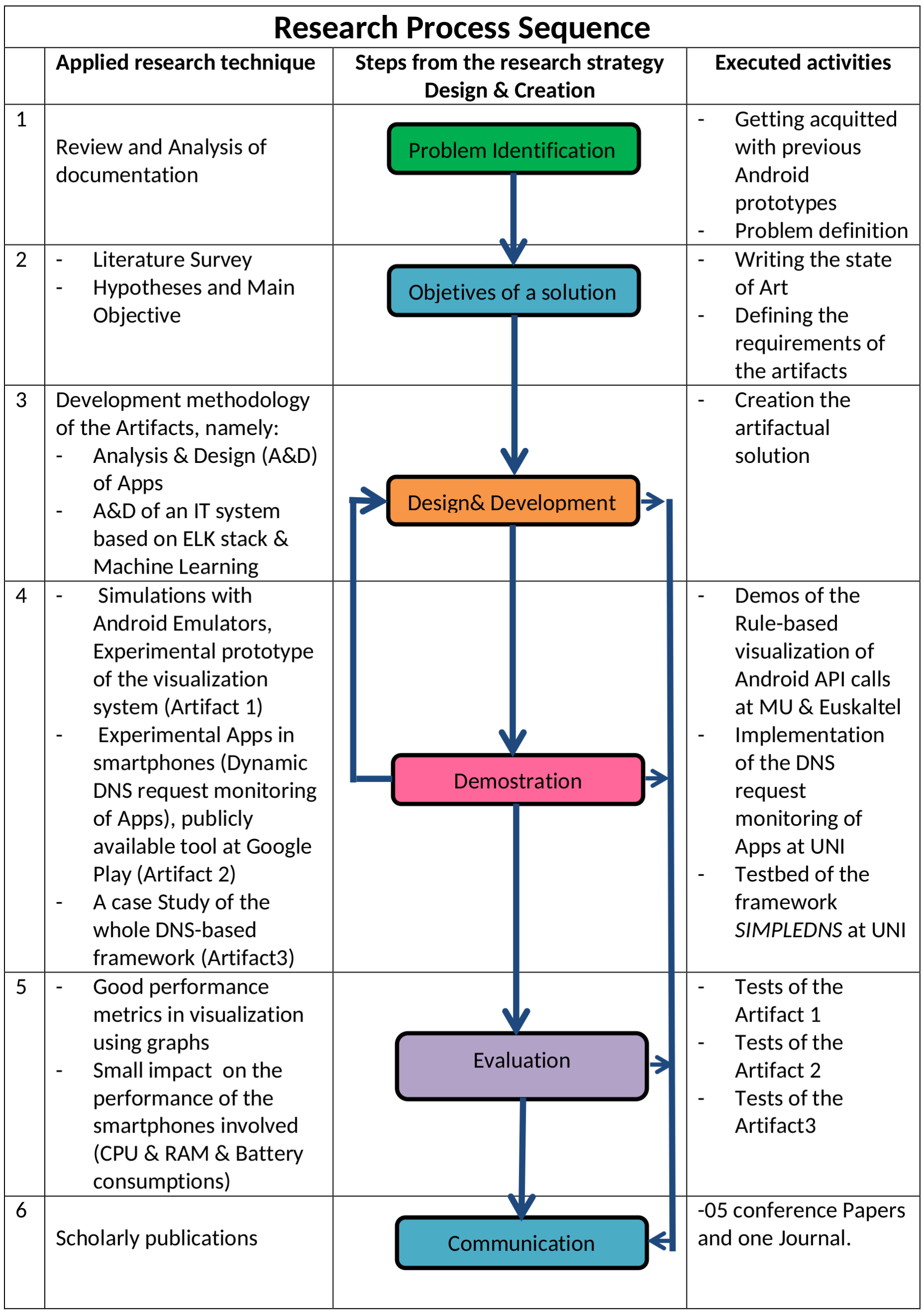
1.6 Thesis Contributions, Papers and Thesis outline

1.6.1 Thesis Contributions

In this thesis we present methods to analyze mobile malware for Android environments when utilizing a behavioral analysis and a collaborative detection framework (smart devices and DNS servers at the operator's infrastructure, working together).

The main contributions of the thesis can be summarized as follows:

1. In [35], we present a survey on dynamic Android malware detection. We first recap the classification and security menaces of mobile malware. Then, we review a number of dynamic malware detection methods proposed in recent years. Additionally, we compare, analyze, and discuss on existing mobile malware detection methods based on certain evaluation criteria. Finally, we summarize open issues in this current research topic and point out future research directions in mobile security, especially on dynamic runtime detection. This has been published in [35].



Research Process Sequence

	Applied research technique	Steps from the research strategy Design & Creation	Executed activities
1	Review and Analysis of documentation	Problem Identification	- Getting acquitted with previous Android prototypes - Problem definition
2	- Literature Survey - Hypotheses and Main Objective	Objetives of a solution	- Writing the state of Art - Defining the requirements of the artifacts
3	Development methodology of the Artifacts, namely: - Analysis & Design (A&D) of Apps - A&D of an IT system based on ELK stack & Machine Learning	Design& Development	- Creation the artifactual solution
4	- Simulations with Android Emulators, Experimental prototype of the visualization system (Artifact 1) - Experimental Apps in smartphones (Dynamic DNS request monitoring of Apps), publicly available tool at Google Play (Artifact 2) - A case Study of the whole DNS-based framework (Artifact3)	Demostration	- Demos of the Rule-based visualization of Android API calls at MU & Euskaltel - Implementation of the DNS request monitoring of Apps at UNI - Testbed of the framework *SIMPLEDNS* at UNI
5	- Good performance metrics in visualization using graphs - Small impact on the performance of the smartphones involved (CPU & RAM & Battery consumptions)	Evaluation	- Tests of the Artifact 1 - Tests of the Artifact 2 - Tests of the Artifact3
6	Scholarly publications	Communication	-05 conference Papers and one Journal.

Figure 1.11: The Design science research process applied to this thesis work, [56].

2. We proposed a visual analysis of the behavior of Android Applications by means of using an instrumentation framework in order to monitor the function calls invoked by App and represent the outcomes using graph modeling. This has been published in [59].

3. Herein we extended the work in [59] in order to allow Android malware de-

tection by using rules-based expert systems. Alternatively, the visual analysis is generated via behavior-related dendrograms out of the traces, that are collected by a remote server from the instrumented Apps. A dendrogram consists of many U-shaped nodes-lines that connect data of the Android application (e.g., the package name of the application, Java classes, and methods and functions invoked) in a hierarchical tree. This has been published in [60].

4. We investigated and proposed infrastructure for monitoring the Android applications in a platform-independent manner, introducing the capture and interception of the URLs requested by the Apps at the smartphone (App traces) [58] at runtime. These traces are collected at a central server where the received URLs are utilized to do string pattern matching with the DNS server records, in this way, in most of the time, we can pinpoint other smart devices doing similar DNS consultations. Initially, the classification of these domains could be done through the usage of blacklists. This has been published in [61]. Later on, in Chapter 4 of this thesis, we have extended the classification of the located DNS records by machine learning..

1.6.2 Papers

Most of these contributions have been published in the following list of publications:

- Oscar Somarriba and Henry Jaentschke, "Dynamic Android Malware Detection: A Survey". In Proc. of the IEEE LATINCOM Workshop 2017, Guatemala City. Guatemala. Nov. 2017 [35].

- Oscar Somarriba, Ignacio Arenaza Nuño, Roberto Uribeetxeberria, and Urko Zurutuza, "Analisis Visual del Comportamiento de Aplicaciones para Android". RECSI XIII. Alicante, Spain (in Spanish). Sept. 2014 [59].

- Oscar Somarriba, Urko Zurutuza, Roberto Uribeetxeberria, Laurent Delosières, and Simin Nadjm-Tehrani, "Detection and Visualization of Android Malware Behavior". Journal of Electrical and Computer Engineering, 2016 [60].

- Oscar Somarriba and Urko Zurutuza, "A Collaborative Framework for Android Malware Detection using DNS & Dynamic Analysis". In Proc. of the 2017 IEEE 37th Central America and Panama Convention (CONCAPAN XXXVII). Managua, Nicaragua. Nov. 2017 [61].

- Oscar Somarriba, "Detecting blacklisted URLs from unmodified and non-rooted Android devices". In Proc. of the 2017 IEEE 37th Central America and Panama Convention (CONCAPAN XXXVII). Managua, Nicaragua. Nov. 2017 [57].

- Oscar Somarriba, Luis Carlos Pérez Ramos, Urko Zurutuza, and Roberto Uribeetxeberria, "Dynamic DNS Request Monitoring of Android Applications via networking". In Proc. of the 2018 IEEE 38th Central America and Panama Convention (CONCAPAN XXXVIII). San Salvador, El Salvador. Nov. 2018 [58].

1.6.3 Thesis Outline

The thesis is written as a monograph composed of five chapters and it is organized as follows.

Chapter 2, **Rule-based visualization of Android API calls**, proposes a working framework to examine Android applications by means of instrumentation and collected traces in a remote server in order to conduct a dynamic analysis and its results are shown in graphs and dendrograms.

Chapter 3, **DNS request monitoring of Android malware**, presents two novel methods to detect Android malware based on capture and interception of URLs (e.g., DNS queries to malign remote servers) combined with dynamic analysis.

Chapter 4, **A Framework to detect Android malware from DNS servers**, presents a framework that provides a detection method that can be used for identifying potentially compromised mobile clients based on DNS traffic analysis. The malware detection is conducted by using DNS-service network traffic (DNS server records) at the operator's infrastructure and dynamic analysis at the host and network level of the smartphones (Apps traces). The main goal of this framework is to combine two approaches: top-down detection by identifying malware domains using DNS-service network traffic and bottom-up detection using the runtime dynamic analysis malware detection on a large number of Apps to hunt down the malware. String pattern matching algorithms and theirs performance evaluations are used within this approach combined with machine learning to classify the name domains.

Finally, Chapter 5, **Conclusions and Future Work**, discusses the results and some remarking conclusions as well as further future research work.

Chapter 2

Rule-based Visualization of Android API calls

This chapter proposes a framework named *AppsShaper* that allows rule-based visualization of Android API calls. The main motivation is as follows. Malware analysts still need to manually inspect malicious software samples that are considered suspicious by heuristic rules. They dissect software pieces and look for malware evidence in the code. The increasing number of malware samples targeting Android devices puts up the demand for analyzing them to find where the malcode is triggered when user interacts with them. In this chapter, a framework to monitor and visualize Android applications' anomalous function calls is described. Our approach includes platform-independent application instrumentation, introducing hooks in order to trace restricted API functions used at run time of the application. These function calls are collected at a central server where the application behavior filtering and a visualization takes place. This can help Android malware analysts in visually inspecting what the application under study does, easily identifying such malicious functions.

2.1 Introduction

As was stated in Chapter 1, mobile malware is one of the greatest menaces in computer security to our networked society.

In order to design a detection and visualization system of Android malicious behavior, we need to characterize Apps. To do that, it is necessary gathering a large amount of data emitted by Apps for smartphones which is crucial for making statistics about the applications' featuring usage or characterizing the applications. However, this is not straightforward, due to Android platforms put restrictions avoiding collecting traces without modifying the firmware or rooting the phone, altogether it is not suitable to deploy on a large scale. To overcome this we have the pressing need to apply another approaches such as instrumenting an application. So, we resort to doing code injection (usually dubbed as well as hooking) into selected API calls that allow to gather the data issued by Apps.

A hook is a functionality provided by software for users of that software to have their own code called under certain circumstances. That code can augment or replace the current code altering the behavior of an Operating Systems (OS), of applications, or of other software components by intercepting function calls or messages or events passed between software components. Code that handles such intercepted function calls, events or messages is called a hook.

This chapter proposes an infrastructure for inserting hooks, collecting the application traces using the hooks, and uploading the traces to a remote server for observation and analysis. To the best of our knowledge, this was one of the first approaches which enable to monitor Android applications at a large scale at the application layer independently from the hardware, and without requiring changes to the firmware or rooting the phone. We also give a detailed description of the infrastructure all the way from the client end elements to the elements that reside at the remote server. Furthermore, we show how our potential changes at the application layer to achieve the mentioned remote monitoring may affect the application performance, by measuring its overhead at the smartphone side. Our monitoring system is portable between smart phones running on Android platforms. As opposed to other works [62], [13], we do not root the smart phone or change its firmware in order to monitor smart phones. The monitoring infrastructure can be made freely available to the Android security community, or adapted for other purposes.

The architecture of the proposed framework can be split up in two flavors regarding the visualization component. First of all, the variant number one, when Android applications are executed, they call a set of functions that are either defined by the developer of the application, or are part of the Android API. Our approach is based on monitoring a desired subset of the functions (i.e., hooked functions) called by the application and then uploading them to a remote server. For this, we use four components: (i) the embedded client, (ii) the Sink on the smart phone side, (iii) the Web Service on the remote server, (iv) and the visualization system based on graphs generated using the tool named Neo4j[1]. Thus

[1]https://neo4j.com/

malicious behavior could be highlighted in the graphs based on a predefined set of anomaly rules. An overview of the framework based on graphs is depicted in Figure 2.1. See further details in Section 2.3

Furthermore, the second variant of the proposed framework focuses on monitoring Android applications' suspicious behavior at runtime, and visualizing (in a slightly different manner) its malicious functions to understand the intention behind them. We also propose a platform-independent behavior monitoring infrastructure composed of four elements: (i) an Android application that guides the user in selecting, instrumenting and monitoring of the application to be examined, (ii) an embedded client that is inserted in each application to be monitored, (iii) a cloud service that collects the application to be instrumented and also the traces related to the function calls, (iv) and finally a visualization component that generates behavior-related dendrograms as well graphs out of the traces. A dendrogram [39] consists of many U-shaped nodes-lines that connect data of the Android Application (e.g., the package name of the application, Java classes and methods and functions invoked) in a hierarchical tree. As a matter of fact, we are interested in the functions and methods which are frequently seen in malicious code. Thus malicious behavior could be highlighted in the dendrogram based on a predefined set of anomaly rules. An overview of the monitoring system is shown in Figure 2.2. See in more in-depth details of the framework in Section 2.3

Monitoring an application at runtime is essential to understand how it interacts with the device, with key components such as the provided application programming interfaces (APIs). An API specifies how some software components (routines, protocols, and tools) should act when subject to invocations by other components. By tracing and analyzing these interactions, we are able to find out how the applications behave, handle sensitive data and interact with the operating system. In short, Android offers a set of API functions for applications to access protected resources [63].

As stated before, Android applications are presented on an .apk file, the container of the application binary that contains the compiled .dex files (in the case of ART, Dalvik bytecode is stored in an .odex file) and the resource files of the application. The resulting .apk file is signed with a keystore to establish the identity of the author of the application. Besides, to build Android applications, a software developer kit (SDK) is usually available allowing access to APIs of the OS [64]. Note that in this chapter we only consider the use of DVM instead of ART. Additionally, two more components are described in order to clarify the background of this chapter: the android-apktool [65], and the smali/backsmali tools. The android-apktool is generally used to unpack and disassemble Android applications. It is also used to assemble and pack them. It is a tool set for reverse engineering third party Apps that simplifies the process of assembling and disassembling Android binary .apk files into smali .smali files and the application resources to their original form. It includes the smali/baksmali tools, which can decode resources (i.e., .dex files) to nearly original form of the source code and rebuild them after making some modifications (i.e. hooking). This enables all these assembling/disassembling operations to be performed automatically in an easy yet reliable way. We utilize smali/baksmali tools to inserts the hooks into the Apps.

However, it is worth noting that the repackaged Android binary .apk files can only possess the same digital signature if the original keystore is used. Otherwise, the new application will have a completely different digital signature.

The remainder of the chapter is organized as follows. Section 2.2 provides an introduction of the state-of-art of the related work. Next we describe the monitoring and visualization architecture in Section 2.3, while we provide the details of the implementational issues of our system in Section 2.4. Later, in Section 2.5, we evaluate the proposed infrastructure and the obtained results by using 8 malware applications. Limitations and Conclusions are presented in Section 2.6 and Section 2.7, respectively.

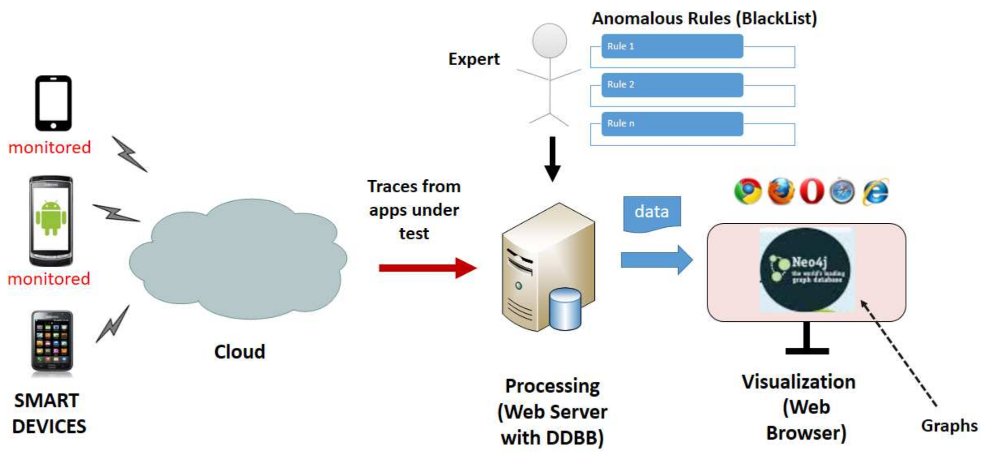

Anomalous Rules (BlackList)
Rule 1
Rule 2
Rule n
Expert
monitored
monitored
SMART DEVICES
Cloud
Traces from apps under test
data
Neo4j
Processing (Web Server with DDBB)
Visualization (Web Browser)
Graphs

Figure 2.1: Overview of the monitoring system with graphs.

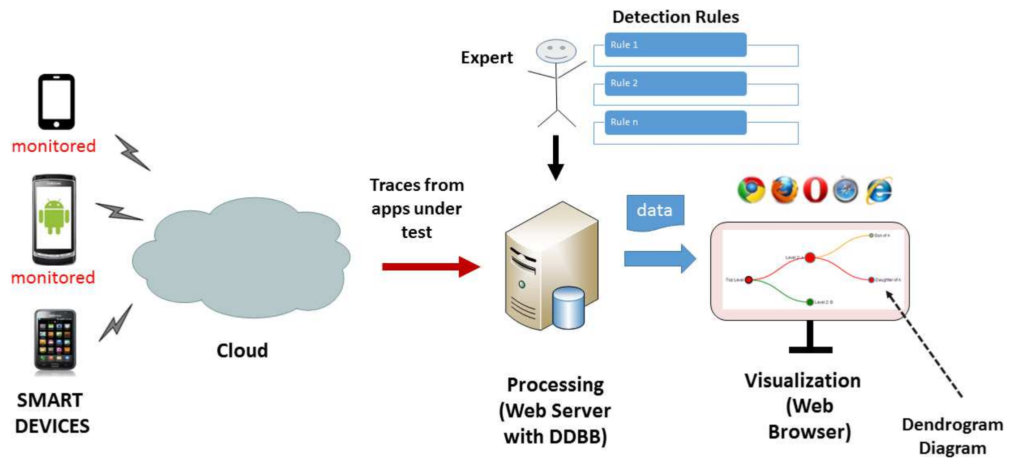


Figure 2.2: Overview of the monitoring system with dendrogram diagrams.

2.2 Related Work

Previous works have addressed the problem of understanding the Android application behavior in several ways. An example of inspection mechanisms for identification of malware applications for Android OS is presented by Karami et al. [66] where they developed a transparent instrumentation system for automating the user interactions to study different functionalities of an App. Additionally, they introduced run-time behavior analysis of an application using input/output (I/O) system calls gathered by the monitored application to within the Linux kernel. Bugiel et al. [27] propose a security framework named *XManDroid* that extends the monitoring mechanism of Android, in order to detect and prevent application-level privilege escalation attacks at runtime based on a given policy. The principal disadvantage of this approach is that the modified framework of Android has to be ported for each of the devices and Android versions in which it is intended to be implemented. Unlike [66] and [27], we profile only at the user level and therefore we do not need to root or to change the framework of Android smart phones if we would like to monitor the network traffic for example.

Other authors have proposed different security techniques regarding permissions in Android applications. For instance, Au et al. [67] present a tool to extract the permission specification from Android OS source code. Unlike the other methods, Dr. Droid and Mr. Hide, implemented by Jeon et al. [68], does not intend to monitor any smart phones. They aim at refining the Android permissions by embedding a module inside each Android application. In other words, they can control the permissions via their module. We also embed a module inside each Android application but it is used to monitor the Android application instead.

In the work by Zhang et al. [63], they have proposed a system called *VetDroid* which can be described as a systematic analysis technique using an App's permission use. By using real-world malware, they identify the callsites where the App requests sensitive resources and how the obtained permission resources are subsequently utilized by the App. To do that, *VetDroid* intercepts all the calls to the Android API, and synchronously monitors permission check information from Android permission enforcement system. In this way, it manages to reconstruct the malicious (permission use) behaviors of the malicious code and to generate a more accurate permission mapping than PScout [67]. Briefly this system [63] applies dynamic taint analysis to identify malware. Different from *VetDroid*, we do not need to root or jailbreak the phone nor do we conduct the permission-use approach for monitoring the smartphone.

Furthermore, in [28], a learning-based method is proposed for the detection of malware that analyzes applications automatically. This approach combines static analysis with an explicit feature map inspired by a linear-time graph kernel to represent Android applications based on their function call graphs. Also, Drebin [29] combines concepts from broad static analysis (gathering as many features of an application as possible) and machine learning. These features are embedded in a joint vector space, so typical patterns indicative of malware can be automatically identified in a lightweight App installed in the smart device. Shabtai et al. [69] presented a system for mobile malware detection that takes into account the analysis of deviations in application networks behavior (App's network traffic patterns). This approach tackles the challenge of the detection of an emerging type of malware with self-updating capabilities based on runtime malware detector (anomaly-detection system) and it is also stand-alone monitoring application for smart devices.

Considering that [28] and Drebin [29] utilize static methods, they suffer from the inherent limitations of static code analysis (e.g., obfuscation techniques, junk code to evade successful decompilation). In the first case, their malware detection is based upon the structural similarity of static call graphs that are processed over approximations, while our method relies upon real functions calls, that can be filtered later on. In the case of Debrin, transformation attacks that are non-detectable by static analysis, as for example based on reflection and bytecode encryption, can hinder an accurate detection.

Although in [69], we have a detection system that continuously monitors App executions. There is a concern about efficiency of the detection algorithm used by this system. Unfortunately, in this case, they could not evaluate the Features Extractor and the aggregation processes' impact on the mobile phone resources, due to the fact that an extended list of features was taken into account. To further enhance the system's performance, it is necessary to retain only the most effective features in such a way that the runtime malware detector system yields relatively low overhead on the mobile phone resources.

In [70], they proposed a two-step malicious App detection method, which combines static and dynamic analysis approaches. During the static analysis, permission combination matrix (the values of the matrix can be used to determine whether some risky combination of two permissions exists) is used to determine whether an App has potential risks. And then those suspicious applications are further sent into the dynamic monitoring module to track the call information of the sensitive APIs while it is running. In other words, for those suspicious Apps, based on the reverse engineering, embed monitoring smali code for those sensitive APIs is done such as sending SMS, accessing user location, device ID, phone number, etc. The monitoring report from the dynamic module is combined with DDMS (Dalvik Debugger Monitor Server) logs, so they can conduct more in-depth manual analysis related with the user privacy information leakage. Our approach in this chapter is similar to this work in particular in the dynamic analysis, however the visualization component is more advanced in our case, and we also run our apps in real smart devices instead of only in the Android emulators in [70]. Moreover, our approach is done in a client/server architecture running on a web service which is flexible, whereas their proposed method is confined to smartphones and we they dont mention the overhead of their monitoring App.

Our proposed infrastructure is related to the approaches mentioned above and employs similar features for identifying malicious applications, such as permissions, network addresses, API calls, and function call graphs. However, it differs in three central aspects from previous work: First, we have a runtime malware detection (dynamic analysis) but abstain from crafting detection in protected environment as the dynamic inspections done by *VetDroid*. While this system provides detailed information about the behavior of applications, they are technically too involved to be deployed on smartphones and detect malicious software directly. Second, our visual analysis system is based on accurate API call graphs, which enables us to inspect directly the App in an easy-to-follow manner in the cloud.

Third, we are able to monitor not just the network traffic, but most of the Restricted and Suspicious API calls in Android. Our platform is more dynamic and simpler than other approaches mentioned above.

Malware in smart devices still poses many challenges and in different occasions, a tool for monitoring applications at a large scale might be required. Given the different versions of Android OS, and with a rising number of device firmwares, modifying each of the devices might become a nontrivial task. This is the scenario in which the proposed infrastructure in this paper best fits. The core contribution of this work is the development of a monitoring and instrumentation system that allows a visual analysis of the behavior of Android applications for any device on which an instrumented application can run. In particular, our work results in a set of graphs/dendrograms that visually render existing API calls invoked by Android malware application, by using dynamic inspection during a given time interval, and visually highlighting the suspicious ones. Consequently, we aim to fill the void of visual security tools which are easy-to-follow, and design for Android environments in the technical literature.

2.3 Platform Architecture

Web Services extend the World Wide Web infrastructure to provide the means for software to connect to other software applications [71]. RESTFul Web Services are Web Services that use the principles of REpresentational State Transfer (REST) [72]. In other words, they expose resources to clients that can be accessed through the Hypertext Transfer Protocol (HTTP) protocol.

When Android applications are executed, they call a set of functions that are either defined by the developer of the application, or are part of the Android API. Our approach is based on monitoring a desired subset of the functions (i.e., hooked functions) called by the application and then uploading information related to their usage to a remote server. The hooked function traces are then represented in a graph structure, and a set of rules are applied to color the graphs in order to visualize functions that match known malicious behavior.

For this, we use four components: the *Embedded client* and the *Sink* on the smart phone side, and the *Web Service* and the *Visualization component* on the remote server side.

A work flow depicting the main elements of the involved systems are shown in Figure 2.3 and Figure 2.4. The main difference between both systems is related with the Visualization component, i.e., in Figure 2.3 it is a rule-based visualization with graph diagrams [59] and in Figure 2.4 it is a rule-based visualization with dendrogram diagrams [60]. The latest approach was taken in order to improve the visualization interpretation that it was a little bit more complicated to do with the Neo4j graphs in many cases. Let us describe the platform architecture in further details.

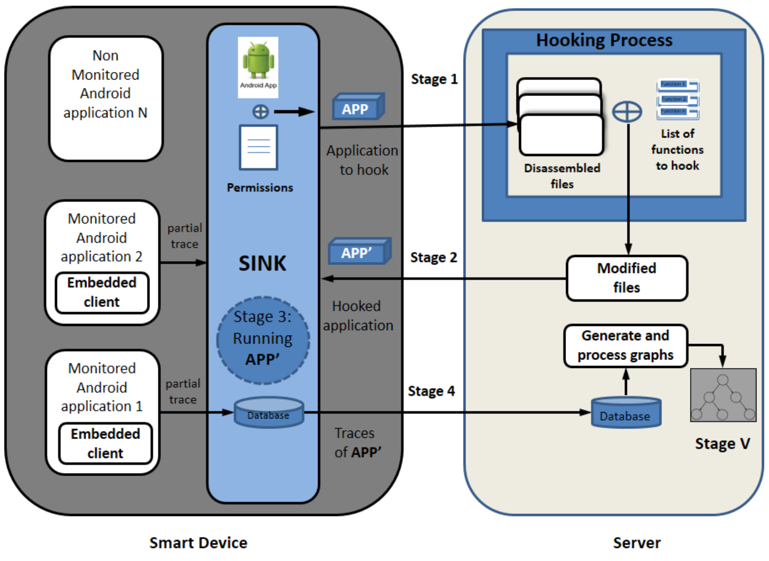


Figure 2.3: Schematics and logical stages of the system with graph diagrams.

In Stage 1, the application under study and a set of permissions aiming to monitor are sent to the Web Service. Next, the main processing task of Stage 2, labeled as Hooking Process, is introduced. In this case, hooks or logging code are inserted in the functions that require at least one of the permissions specified at the previous Stage. The new "augmented" application will be referred to as APP' from now on. Stages 3, 4, and 5 consist of running APP', saving the traces generated by APP' in the server's database and showing the results as visualization graphs/dendrograms, respectively. The aforementioned infrastructure for platform-independent monitoring of Android applications is aimed to provide behavioral analysis without modifying the Android OS or root access to the smart device.

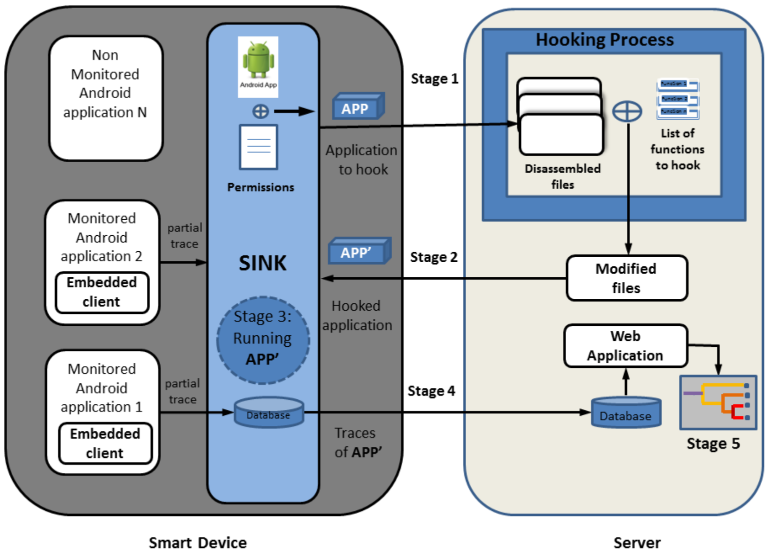


Figure 2.4: Schematics and logical stages of the system with dendrogram diagrams.

2.3.1 Embedded client and Sink

The monitoring system consists of two elements: an embedded client that will be inserted into each application to be monitored, and a Sink that will collect the hooked functions that have been called by the monitored applications. The embedded client simply consists of a communication module that uses the User Datagram Protocol (UDP) for forwarding the hooked functions to the Sink. Here, JavaScript Object Notation (JSON) is used when sending the data to the Sink, which allows sending dynamic data structures. In order to know the origin of a hooked function that has been received by the Sink, the corresponding monitored application adds its application hash, its package name, and its application name to the hooked function which we call a partial trace before sending it to the Sink.

The partial traces are built by the prologue functions (i.e., hook functions), that are placed just before their hooked functions, and which modify the control flow of the monitored applications in order to build the partial traces corresponding their hooked functions and passing the partial traces as parameter to the embedded client. Only the partial traces are built by the monitored application so that we add little extra overhead to the monitored application. The insertion of the embedded client and of the prologue functions in the Android application to monitor is explained in Section 2.3.3.

The embedded client is written using the smali syntax and is included on each of the monitored applications at the Web Service, at the same time that the functions hooks are inserted, before the application is packed back into an Android binary .apk file.

The Sink, on the other hand, is implemented as an Android application for portability both as a service and an activity whose service is started at the boot time. It is responsible for receiving the partial traces issued from all the monitored applications clients via a UDP socket, augmenting the partial traces to get a trace (i.e. adding a timestamp and the hash of the ID of the phone), storing them, and sending them over the network to the Web Service. As for the activity, it is responsible for managing the monitored applications via a UI, sending the applications to hook to the Web Service, and downloading the hooked applications from the Web Service. By hooked applications, we mean the applications in which hooks have been inserted. Once an application has been hooked then we can monitor it.

Before storing the traces in a local database, the Sink first stores them in a circular buffer which can contain up to 500 traces. The traces are flushed to the local database when any of the following conditions are met: (i) the buffer is half full, (ii) the Sink service is shutting down, or (iii) upon an activated timeout expiring. This bulk flushing enables the Sink to persist the traces more efficiently. Unfortunately, if the service is stopped by force, we lose the traces that are present in the circular buffer. Once the traces are persisted in the local database, the time-out is rescheduled. Every hour, the Sink application tries to send the traces persisted in the local database out to the Web Service. A trace is removed locally upon receiving an acknowledgment from the Web Service. An acknowledgment is issued when the Web Service has been able to record the trace in a SQL database with success. If the client cannot connect to the Web Service, it will try again at the next round.

When a user wants to monitor an application, a message with the package name as payload is sent to the Sink service which keeps track of all the applications to monitor in a list. When a user wants to stop monitoring a given application, a message is sent to the Sink service which removes it from its list of applications to monitor.

2.3.2 The Web Service

This server provides the following services to *Sink*: upload applications, download the modified applications and send the traces. Now the key part of the whole system where the logic of the method presented lies, is the tool that implements the application, a process known as "hooking". In the following, we explain it. The Web Service, implemented as a Servlet on a Tomcat web application server, is a RESTful Web Service which exposes services to clients (e.g., Android smart phone) via resources. The Web Service exposes three resources which are three code pages enabling the Sink to upload an application to hook, download a hooked application, and send traces. The hooking process is explained in more detail in Section 2.3.3.1.

The file upload service allows the Sink to send the target application to monitor, and triggers the command to insert all the required hooks and the embedded client to the application. Also, it is in charge of storing the submitted Android binary .apk file on the server and receiving a list of permissions. This set of permissions will limit the amount of hooks to monitor, hooking only the API function calls linked to these permissions. Conversely, the file download service allows the Sink to download the previously sent application, which is now prepared to be monitored. A ticket system is utilized in order to keep tracking of the current application under monitoring. The trace upstream service allows the Sink to upload the traces stored on the device to the server database and remove the traces from the devices local SQLite database. Upon receiving traces, the Web Service persists them in a SQL database and sends an acknowledgment back to the Sink. In case of failure in the server side or in the communication channel, the trace is kept locally in the SQLite database until the trace is stored in the server and an acknowledgment is received by the Sink. In both cases, it might occur that the trace has just been inserted in the SQL database and no answer is sent back. Then the Sink would send again the same trace and we would get a duplication of traces. However, the mechanism of primary key implemented in the SQL database prevents the duplication of traces. A primary key is composed of one or more data attributes whose combination of values must be unique for each data entry in the database. When two traces contain the same primary key, only one trace is inserted while the insertion of the other one throws an exception. When such an exception is thrown, the Web Service sends back an acknowledgment to the Sink so as to avoid the Sink to resend the same trace (i.e. force the Sink to remove from its local database the trace that has already been received by the Web Service).

2.3.3 Instrumenting an Application

In this Section, we first describe the process of inserting hooks into an Android application and then we show an example of a hook implementation. A tutorial on instrumentation Android applications is presented by Arzt et al. in [73].

However, before proceeding with the insertion of instrumentation code to the decompiled APK below, we would like to clarify the effect of disassembling the uploaded applications, i.e., the differences between the original code and code generated after instrumentation. Briefly, the disassembling of the uploaded application is performed by using the smali/baksmali tool which is assembler/disassembler respectively for the dex format [2]. This is the format used by Dalvik, one of the Android's JVM implementations. Thus, the disassembling is able to recover an assembler-like representation of the Java original code. This representation is not the original Java source code (Baksmali is a disassembler, not a decompiler after all). However it creates both an exact replica of the original binary code behavior, and high-level enough to be able to manipulate it in an easy way. This is why we can add additional instructions to instrument the original code for our purposes and then re-assemble it back to a dex file that can be executed by Android's JVM. On the other hand, as discussed in [73], instrumentation of applications outperforms static analysis approaches, as instrumentation code runs as part of the target App, having full access to the run-time state. So, this explains the rationale behind introducing hooks in order to trace core sensitive or restricted API functions used at run time of the Apps. In other words, the smali code reveals the main restricted APIs utilized by the Apps under test, even in the presence of source code obfuscation. We can therefore resort to monitoring these restricted APIs and keep tracking of those Android suspicious programs' behavior

2.3.3.1 Hooks insertion

The hooking process is done in 6 steps: (i) receiving the application to hook from the smart phone, (ii) unpacking the application and disassembling its Dalvik byte code via the Android apktool, (iii) modifying the application files, (iv) assembling Dalvik byte code and packing the hooked application via the Android apktool, (v) signing the hooked application, and (vi) sending the hooked application upon request of the smart phone.

Step iii can be subdivided into several sub-steps:

[2]https://source.android.com/devices/tech/dalvik/dex-format.html

1. adding the INTERNET permission in the *AndroidManifest* to enable the embedded client inserted in the application to hook to communicate with the Sink via UDP sockets.

2. parsing the code files and adding invocation instructions to the prologue functions (PF) before their corresponding hooked functions. When the monitored application is running, before calling the hooked function, its corresponding prologue function will be called and will build its corresponding partial trace. The list of desired functions to hook is provided by the administrator of the Web Service. For instance, if the administrator is interested in knowing the applications usage, it will hook the functions that are called by the application when starting and when closing.

3. adding a class that defines the prologue functions. It is worth noting that there will be as many prologue functions as functions to hook. Each prologue function builds its partial trace. Since we do not log the arguments of the hooked functions, the partial traces that are issued by the same monitored application, will only differ by the name of the hooked function. It is also worth noting that the prologue functions are generated automatically.

Since every Android application must be signed by a certificate for being installed on the Android platform, we use the same certificate to check if the hooked application comes from our Web Service. For this, the certificate used in the Web Service has been embedded in the Sink application. This prevents attackers from injecting malicious applications by using a man-in-the-middle attack between the smart phone and the Web Service.

2.3.3.2 Hook example

Consider a case where the function *sendTextMessage*, used to send short messages (SMS) on the Android platform, is to be logged in a monitored application. This function is called in the main activity class of the application corresponding to the code Listing 2.1. As for the class shown in Listing 2.2, it defines the prologue functions and the function responsible for passing the partial traces, built by the prologue functions, to the embedded client. For space reasons, we will not show the embedded client.

In the main activity class corresponding to the class shown in Listing 2.1, the function *sendTextMessage* is called at line 4 with its PF *log_sendTextMessage* which has been placed just before at line 3. Since the hooked function may modify common registers used for storing the parameters of the hooked function and for returning objects, we have preferred placing the prologue functions before their hooked functions. The register *v1* is the object of the class *SmSManager* needed to call the hooked function. As for the registers *v2* to *v6*, they are used for storing the parameters of the hooked function. Since our prologue functions (PFs) are declared as static, we can call them without instantiating their class 2, and therefore we do not need to use the register *v1*.

An example of the Monitor log class is shown in Listing 2.2. The name of the class is declared at line 1. At lines 3 and 10, two functions are defined, namely *log_sendTextMessage* and *sendLog*. The former function, prologue function of the hooked function *sendTextMessage*, defines a constant string object containing the partial trace at line 5 and puts it into the register *v0*. Then the function *sendLog* is called at line 6 with the partial trace as parameter. The latter function saves the partial trace contained in the parameter *p0* into the register *v0* at line 13. At line 15 and 16, two new instances are created respectively: a new thread and new instance of the class *EmbeddedClient*. Their instances are initialized respectively at lines 17 and 18. Finally, the thread is started at line 19 and the partial trace is sent to the Sink. It is worth noting that in these two examples, we have omitted some elements of the code which are replaced by dots to facilitate the reading of the code.

Listing 2.1: Main activity class

```
1  .class public Lcom/mainactivity/MainActivity;
2  ...
3  invoke-static/range {v2 .. v6}, log_sendTextMessage(...)
4  invoke-virtual/range {v1 .. v6}, sendTextMessage(...)
5  ...
```

Listing 2.2: Monitor log class

```
1  .class public Lorg/test/MonitorLog;
2  ...
3  .method public static log_sendTextMessage(...)
4  ...
5  const-string v0, "packageName: com.testprivacy, ..."
6  invoke-static {v0}, sendLog(Ljava/lang/String;)
7  return-void
8  .end method
9
10 .method public static sendLog(Ljava/lang/String;)
```

```
.locals 3
.parameter payload
move-object v0, p0
...
new-instance v1, Ljava/lang/Thread;
new-instance v2, Lorg/test/EmbeddedClient;
invoke-direct {v2, v0}, init(Ljava/lang/String;)
invoke-direct {v1, v2}, init(Ljava/lang/Runnable;)
invoke-virtual {v1}, start()
return-void
.end method
```

2.3.4 Visualization

The visualization of anomalous behavior is the last component of the proposed architecture. Again, here we have two cases: one with Neo4j graphs and a second approach with dendrogram diagrams.

Let us consider the first approach. In order to perform a visual analysis of the behavior of the applications, a graph-based NoSQL database, Neo4j, is used in Figure 2.3. Neo4j stores the data in a graph-oriented structure, instead of using the relational tables of conventional databases.

The graphs are elaborated through relations of type "an Application includes several Classes that in turn call Functions". The first top node, "Application", contains the name of the application package, which is unique for each of the existing applications, while the second node, "Class", represents the name of the application component. Android that has called the "API call", the "Function" node. Once the information collected by the Web service in the database is obtained, its entire function call structure can be filtered and treated. Initially a graph is generated without including colors using the Cypher Query Language from Neo4j that allows to operate and apply transformations in the graph. To do this, an expert must complete and chain a set of rules that help highlight the known malicious behavior (for example, the call to the *SendMessageText* function). For this, the nodes are searched in the lower part of the graph related to calls to API functions considered malicious. The rules include the search and representation of malicious functions (represented in red), suspicious or malicious but not critical (in orange) and benign (in green). When a malicious behavior is detected, such as sending SMS messages to a premium payment number without the user's consent, the node under red inspection will be represented as a warning signal using Cypher.

Alternatively, we consider the second case regarding the visualization component as shown in Figure 2.4. In order to perform a visual analysis of the applications' behavior in a simplified way, a D3.js (or just D3 for Data-Driven Documents[3]) graph was used. D3 is an interactive and a browser-based data visualizations library to build from simple bar charts to complex infographics. In this case, it stores and deploys graph oriented data on a tree-like structure named dendrograms using conventional database tables. Generally speaking, a graph visualization is a representation of a set of nodes and the relationships between them shown by links, (*vertices* and *edges*, respectively).

This way, we are able to represent each of the analyzed application's behavior with a simple yet illustrative representation. In general, the graphs are drawn according to the schema depicted in Fig. 2.5. The first left-hand (root) node, "Application", contains the package name of the application, which is unique to each of the existing applications. The second middle node (parent), "Class", represents the name of the Android component that has called the API call. The third node, "Function" (the right-hand or child node), represents the names of functions and methods invoked by the application. It is worth noting that each application can include several classes and each class can call various functions or methods.

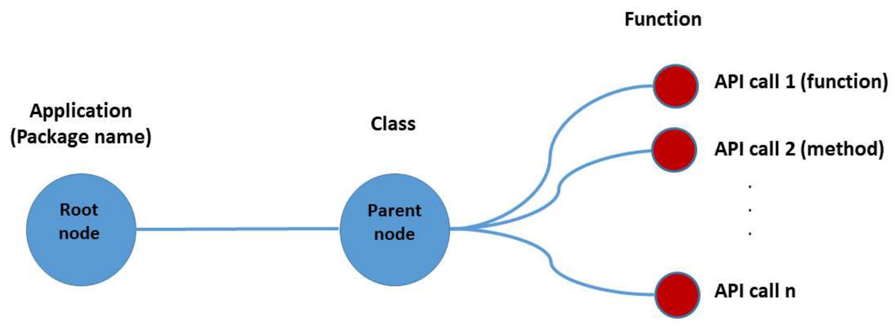


Figure 2.5: Schema used for the graphs/dendrograms.

[3]JavaScript library available at d3.org

In other words, function calls are located in the right-hand side of the graps/dendrogram. For each node at this depth we are looking for known suspicious functions derived from a set predefined of rules as described below.

2.3.4.1 *Rules "Generation"*

The rules aim to highlight restricted API calls, which allow access to sensitive data or resources of the smartphone and are frequently found in malware samples. These could be derived from the static analysis where the classes.dex file is converted to smali format, as mentioned before, to get information considering functions and methods invoked by the application under test. On the other hand, it is well-know that many types of malicious behaviors can be observed during runtime only. For this reason we utilize dynamic analysis, i.e., Android applications are executed on the proposed infrastructure (see Figures 2.3 and 2.4) and interact with them. As a matter of fact, we are only interested to observe the Java based calls, which are mainly for runtime activities of the applications. This includes data accessed by the application, location of the user, data written to the files, phone calls, sending SMS/MMS, data sent and received to or from the networks, etc.

For the case that an application requires user interactions, we resort to do that manually so far. Alternatively, for this purpose one can use MonkeyRunner toolkit, which is available in Android SDK.

In [29] and in [74], authors list API functions calls that grant access to restricted data or sensible resources of the smartphone, which are very often seen in malicious code. We base our detection rules in those suspicious APIS calls. In particular, we use the following types of suspicious APIs:

- API calls for accessing sensitive data, e.g IMEI and USIMnumbeleakage, such as *getDeviceId()*, *getSimSerialNumber()*, *getImei()* & *getSubscriberId()*
- API calls for communicating over the network, for example *setWifiEnabled()* & *execHttpRequest()*
- API calls for sending and receiving SMS/MMS messages, such as *sendTextMessage()*, *SendBroadcast()* and *sendDataMessage()*
- API calls for location leakage, such as *getLastKnownLocation()* and *getLatitude()*, *getLongitude()*, and *requestLocationUpdates()*

- API function calls for execution of external or particular commands like *Runtime.exec()*, and *Ljava/lang/Runtime;->exec()*
- API calls frequently used for obfuscation and loading of code, such as *DexClassLoader.Loadclass()* and *Cipher.getInstance()*

Here the rule module uses the above mentioned API calls to classify the functions and methods invoked on the runtime of the applications into three classes, i.e., Benign, Adware or Malware. So in this way, we can generate IF-THEN rules (cf. rules-based expert systems). Next we show example rules that describe suspicious behavior. Some of the rules generated by us are similar or resemble the ones in [75], as follows, namely:

1. A rule that shows that the examined App is not allowed to get the location of the smart device user:

   IF Not ($ACCESS\_FINE\_LOCATION$) AND $CALL\_getLastKnownLocation$ THEN Malware

2. Another rule might detect that the application is trying to access sensitive data of the smartphone without permission:

   IF Not ($READ\_PHONE\_STATE$) AND $CALL\_getImei$ THEN Malware

Our approach selects from the database those functions that have been executed that match the suspicious functions described in the rules. Package name, and class name of such function are colored accordingly to the "semaphoric" labeling described in Section 2.5.

To illustrate the basic idea we choose a malware sample, known as *FakePlayer*, in order to draw its graph. Thus, by means of running the filtering and visualization operations we end up with the graph of the malware, shown in Figure 2.6.

The system allows adding new rules in order to select and color more families of suspicious functions.

2.4 Testbed and Experimentation

Before introducing the reader into the results of using the monitoring and visualization platform, we need to explain the testbed. We first describe the experiment set up, then we follow the steps of running the client-side Sink.

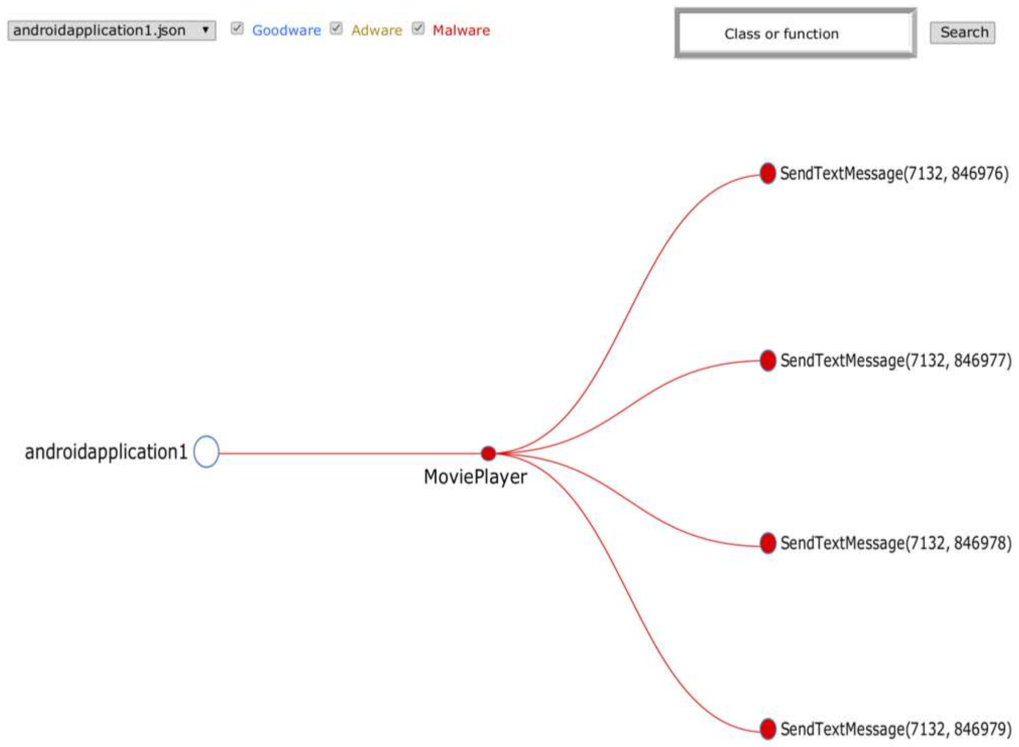

androidapplication1.json
Goodware
Adware
Malware
Class or function
Search
androidapplication1
MoviePlayer
SendTextMessage(7132, 846976)
SendTextMessage(7132, 846977)
SendTextMessage(7132, 846978)
SendTextMessage(7132, 846979)

Figure 2.6: The simplified dendrogram of the malware *FlakePlayer* has been generated using the D3. Note that at the upper left corner of the figure there is a combobox to select the monitored malware (here, for simplicity, we use a shortened version of package name of the App, i.e., *androidapplication1*). Besides, lining up to the right of the combobox, there are three activated checkboxes, labeled as: Goodware in blue, Adware in orange, and Malware in red. Also, at the upper right corner of the figure, there is a **search button** that allows to look for classes or functions. The complete package name of the malware *FakePlayer* is *org.me.androidapplication1.MoviePlayer*.

2.4.1 Experiment set up

All the experiments have been realized on a Samsung Nexus S with Android Ice Cream Sandwich (ICS). The Nexus S has a 1 GHz ARM Cortex A8 based CPU core with a PowerVR SGX 540 GPU, 512 MB of dedicated RAM and 16 GB of NAND memory, partitioned as 1 GB internal storage and 15 GB USB storage.

We have explored different Android applications in order to evaluate the whole framework, some of these samples have been taken from the Android Malware Genome Project[4]):

[4]The Android Malware Genome Project dataset is accessible at `http://www.malgenomeproject.org/`

- **FakePlayer malware.**
- **SMSReplicator malware.**
- **iMatch malware.**
- **DroidKungFu1 malware.**
- **DroidKungfu4 malware.**
- **The spyware GoldDream in two flavors.**
- **GGTracker malware.**

2.4.2 Client-side monitoring

The activities in Figure 2.7(a) display all the applications installed on the device that did not come preinstalled, from which the user selects a target application to monitor. Once an application is selected, the next step is to choose which permission or permissions the user wants to monitor. This can be observed in the third snapshot (white background) of Figure 2.7(c). Following the permissions clearance, the interface guides the user along several activities starting with the uploading of the selected application which is sent to the Web Service where the hooks are inserted. After this hooking process has finished, the modified application is downloaded from the web service. Afterwards, the original application is uninstalled and replaced by the modified application. Finally, a toggle allows to start and stop monitoring the application at any time by the user.

We focus on the functions of the Android API that require, at least, one permission. This allows the user to select from the Sink those permissions that are to be monitored at each application. This allows understanding how and when these applications use the restricted API functions. The PScout [67] tool was used to obtain the list of functions in the "API permission map". This way, the permission map obtained contains (Android 4.2 version API level 17) over thirty thousand unique function calls and around seventy five different permissions. Besides, it is worth mentioning here that we refer to as "restricted API functions" those associated with a sensitive API as well as sensitive data stored on device and privacy-sensitive built-in sensors (GPS, camera, etc.). The first group is any function that might generate a "cost" for the user or the network. These APIs include [64], among others: Telephony, SMS/MMS, Network/Data, In-App Billing, NFC

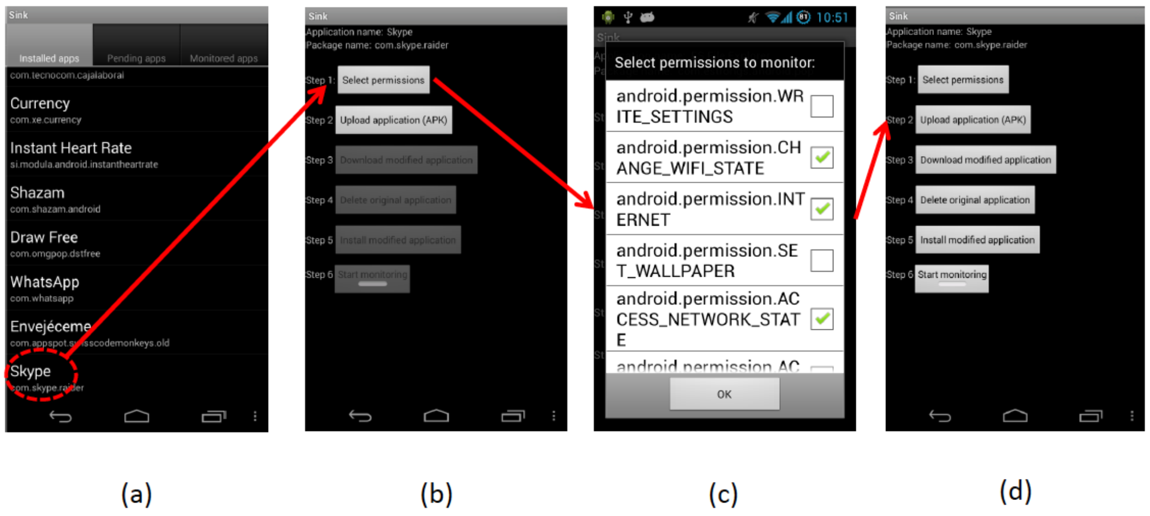

Sink
Installed apps
Pending apps
Monitored apps
com.tecnocom.cajalaboral
Currency
com.xe.currency
Instant Heart Rate
si.modula.android.instantheartrate
Shazam
com.shazam.android
Draw Free
com.omgpop.dstfree
WhatsApp
com.whatsapp
Envejéceme
com.appspot.swisscodemonkeys.old
Skype
com.skype.raider
Application name: Skype
Package name: com.skype.raider
Step 1 Select permissions
Step 2 Upload application (APK)
Step 3 Download modified application
Step 4 Delete original application
Step 5 Install modified application
Step 6 Start monitoring
10:51
Select permissions to monitor:
android.permission.WRITE_SETTINGS
android.permission.CHANGE_WIFI_STATE ✓
android.permission.INTERNET ✓
android.permission.SET_WALLPAPER
android.permission.ACCESS_NETWORK_STATE ✓
OK
(a)
(b)
(c)
(d)

Figure 2.7: User Interface of the Sink. (a) Choosing the Application, (b) Selecting the menu for Permissions, (c) Electing the Permissions, and (d) Steps of the monitoring process.

(Near Field Communication) Access. Thus, by using the API map contained in the server's database, we are able to create a list of restricted ("suspicious") API functions.

The trace managing part is a service that runs in background with no interface, and is in charge of collecting the traces sent from the individual embedded clients, located on each of the monitored applications. It adds a timestamp and the hash of the device ID, and stores them on a common circular buffer. Finally, the traces are stored in bulk on a common local SQLite database, and are periodically sent to the web service and deleted from the local database.

In summary, the required steps to successfully run an Android modified instrumented application are listed in Figure 2.7(d) and comprising the followings:

- Step 1: *Select permissions*. Set up and run the platform. Choose an application APP to be monitored on the device. Elect the permission list.
- Step 2: *Upload the Application (APK)*. Then, when this command is launched to upload the applications to the Web Service, the hooking process is triggered.
- Step 3: *Download modified application*. This starts the downloading of the hooked application.
- Step 4: *Delete original application*. This command starts the uninstallation process of the original application.

- Step 5: *Install modified application*. This command starts the installation process of the modified application using Android's default application installation window.

- Step 6: *Start monitoring*. Finally, a toggle is enabled and can be activated or disabled to start or stop monitoring that application as chosen by the user.

2.5 Results

To evaluate our framework, in this section we show the visualization results for several different applications to both benign and malicious. Then we proceed to evaluate the Sink application in terms of CPU utilization and ratio of partial traces received. Finally, we estimate the CPU utilization of a monitored application and its responsiveness.

2.5.1 Visual analysis of the traces with Neo4j-graphs

As stated above, a set of rules predefined by experts allows us to identify functions "Suspicious" APIs, and depending on their parameters, colors are assigned to them. By doing so, it allows us to quickly identify the functions and associate them with related elements. By applying the classification of functions based on a color for each node of the graph, this allows the construction of a "visual map" that describes and helps the analysis of its operation, see subsection 2.5.2 for the description of the colors used as state of the security threat. In addition, this graph is suitable for guiding the analyst during the examination of the classification of a dangerous malware sample because the red shadow of the nodes indicates malicious structures identified by the monitoring infrastructure. This revision must be made between all the nodes of the functions called at the lowest level of each branch of the tree of the graph. However, in order to completely color the graph of the application until reaching the root node, it is necessary to resort to a bottom-up analysis of the neighborhood of each function invoked and associated. Therefore, if one of the branches of the graph is colored red, then the App is considered potentially malicious.

By means of running the filtering and visualization operations we end up with the graph of the malicious software, shown in Fig. 2.8 and Fig. 2.9. The whole process of obtaining the graph of the API calls of the App under test is described in [59]. Of course, the system allows adding new rules in order to select and color more families of suspicious functions.

In order to give a favor of the what kind of results we can obtain with Neo4j, we illustrate in Figures 2.8 and 2.9; the Neo4j graphs of the Pjapps malware, and the DroidKungFu malware, respectively. Here we can notice that every function calls is denoted by a number making sometimes difficult the conduct the visual analysis so can look for another visual representation technique, which is introduced in the next subsection.

2.5.2 Visual analysis of the traces with dendrogram diagrams

As mentioned before, a set of predefined rules allows us to identify the suspicious API functions" and depending on its parameters (e.g., application attempts to send SMS to a short code that uses premium services) we assign colors to them. This enables us to quickly identify the functions and associate them with related items. On top of that, by applying the color classification of each node of the graph associated to a function in accordance with the color code (gray, orange and red) explained below, allows a "visual map" to be partially constructed. Furthermore, this graph is suitable to guide the analyst during the examination of a sample classified as dangerous because, for example, the red shading of nodes indicates malicious structures identified by the monitoring infrastructure. In particular, to give a flavor of this analysis, the dendrogram of *FakePlayer* in Figure 2.6 provides to the user an indication of the security status of the malware. Different colors indicate the level of alarm associated with the currently analyzed application:

- Gray: indicates that no malicious activity has been detected, as of yet.
- Orange: indicates that no malicious behavior has been detected in its graph, although some Adware may be presented.
- Red: indicates in its graph that a particular application has been diagnosed as anomalous, meaning that it contained one or more "dangerous functions" described in our blacklist. Moreover, it could imply the presence of suspicious API calls such as *sendTextMessage* with forbidden parameters; or the case of using restricted API calls for which the required permissions have not been requested (root exploit).

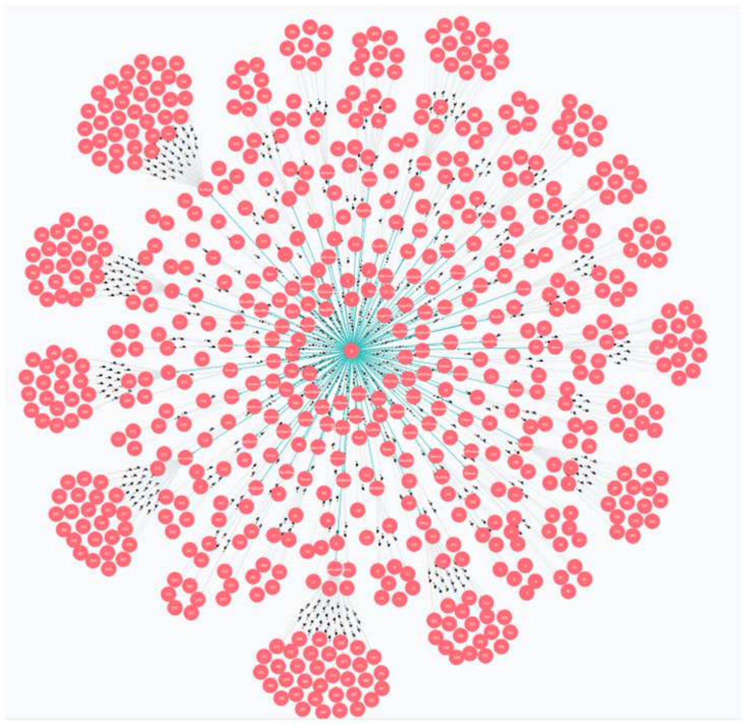


Figure 2.8: Graphs using Neo4j with the malware PjApps.

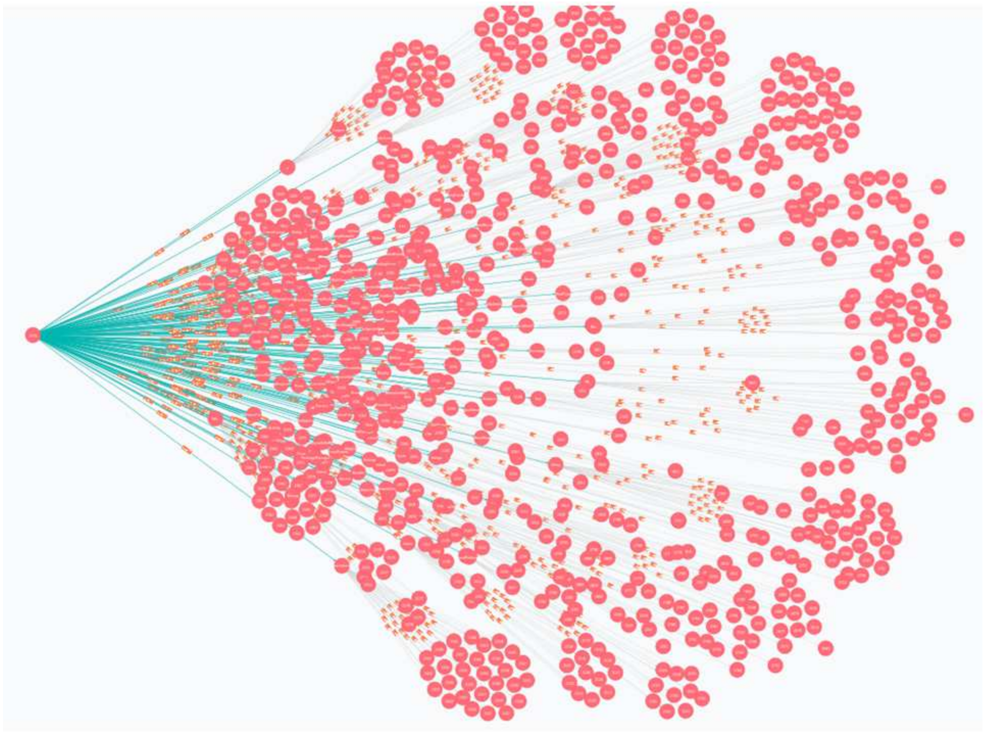


Figure 2.9: Graphs using Neo4j with the malware DroidKungFu.

So, it is possible to conduct a visual analysis of the permissions and function calls invoked per application, where using some kind of "semaphoric labelling" allows to identify easily the benign (in gray and orange colors) applications. For instance in Fig. 2.6 there is a presence of malware, and the nodes are painted in red. The dendrogram shown for *FakePlayer* confirms its sneaky functionality by forwarding all the SMS sent to the device to the previously set phone number remaining unnoticed. For the sake of simplicity, we reduce the API function call *sendTextMessage*(phoneNo, null, SMS Content, null, null) to *sendTextMessage*(phoneNo, SMS Content).

It uses the API functions to send four (see Figure 2.6) premium SMS messages with digit codes on it in a matter of milliseconds. Of course, sending a SMS message does not have to be malicious per se. However, e.g., if this API utilizes numbers less that 9 digits in length, beginning with a "7" combined with SMS messages, this is considered a costly premium-rate service and a malware that sends SMS messages without the user's consent. The malware evaluated sends SMS messages that contain the following strings: 846976, 846977, 846978, and 846979. The message may be sent to a premium SMS short code number "7132", which may charge the user without his/her knowledge.

This implies financial charges. Usually, when this malware is installed, malicious Broadcast Receiver is enrolled directly to broadcast messages from malicious server to the malware, so that user can not understand whether specific messages are delivered or not. This is because the priority of malicious Broadcast Receiver is higher than SMS broadcast receiver. Once the malware is started, sending the function call *sendTextMessage* of SMS Manager API on the service layer, a message with premium number is sent which is shown in Figure 2.6.

2.5.3 Interactive Dendrograms

In general, it is needed to conduct the visual analysis from different perspectives. To do that we have developed an interactive graph visualization [1]. So, we have four options or features in the D3 visualization of the application to monitor, namely: a) selection of full features of the application (Goodware checkbox, Adware checkbox and Malware checkbox), b) the Goodware choice of the App, c) the Adware checkbox of the application, and d) the Malware checkbox to look for malicious code. The analyst can choose to observe a particular java class or function

Table 2.1: **Malware family, detection rules and suspicious functions**

Malware Family	Detection Rules	Suspicious Functions
FakePlayer	IF (SEND_SMS) && (CALL_sendTextMessage() with preset numbers) THEN Malware	sendTextMessage(7132, null, 846976, null, null);
SMSReplicator	IF (SEND_SMS) && (CALL_sendTextMessage() with preset numbers) THEN Malware	sendTextMessage(1245, null, {From: 123456789 Hi how are you}, null, null);
iMatch	IF Not (ACCESS_FINE_LOCATION) && IF (SEND_SMS) THEN Malware	requestLocationUpdates(); sendTextMessage();
DroidKungFu1	[IF (INTERNET) && IF Not (ACCESS_FINE_LOCATION)] || [IF (READ_PHONE_STATE) && IF (INTERNET)] THEN Malware	getLatitude(); getLongitude(); getDeviceid(); getLIne1Number(); getImei();
DroidKungFu4	IF (INTERNET) && IF (READ_PHONE_STATE) THEN Malware	getDeviceid(); getLIne1Number(); getSimSerial(); getImei();
GoldDream (Purman)	[IF (READ_PHONE_STATE) && IF Not (SEND_SMS)] || [IF Not (READ_PHONE_STATE) && IF (INTERNET)] THEN Malware	getDeviceId(); getLIne1Number(); getSimSerial(); sendTextMessage(); getImei();
GoldDream (Dizz)	[IF (READ_PHONE_STATE) && IF Not (SEND_SMS)] || [IF Not (ACCESS_FINE_LOCATION) && IF (INTERNET)] THEN Malware	getDeviceId(); getLIne1Number(); getSimSerial(); sendTextMessage(); requestLocationUpdates(); getImei();
GGTracker	[IF (READ_PHONE_STATE) && Not (SEND_SMS)] || [IF Not (ACCESS_FINE_LOCATION) && IF (INTERNET)] THEN Malware	getDeviceId(); getLIne1Number(); getSimSerial(); sendTextMessage(); requestLocationUpdates(); getImei();

by typing the name of it inside the search box, and clicking on the related Search button.

Figures 2.10 & 2.11 illustrate a big picture of the whole behavioral performance of the malware *DroidKungFu1* whose package name is *com.nineiworks.wordsXGN*, and the malicious function calls invoked. For the sake of simplicity, we shorten the package name of *DroidKungFu1* to *wordsXGN* in the dendrogram. As a matter of fact, we apply a similar labeling policy to the other dendrograms. Moreover, we have the Dendrograms for the *DroidKungFu4* in Figures 2.12. In particular, in the graph of Figure 2.12(a), we conduct the visual inspection by using full features (i.e., all the checkboxes active simultaneously) looking for red lines (presence of malware, if that is the case). Furthermore, in the graph of Figure 2.12(b), now we can focus our visual examination in the malicious functions carried by the application. The visual analysis of the *DroidKungFu1* and *DroidKungFu4* include encrypted root exploits, Command & Control (C& C) servers which in the case of *DroidKungFu1* is in plain text in a Java class file, and shadow payload (embedded App). In Table 2.1, we have some of the suspicious function calls utilized by the malware which pop-up from the dendrograms. Regarding the IF THEN rules, the allowed clauses or statements in our infrastructure are: permissions and API functions calls. The fundamental operators are: Conditional-AND which is denoted by &&, Conditional-OR which is denoted by ||, and Not. For example, If examined App does not have permission to send SMS messages in the *AndroidManifest* file and that App try to send SMS messages with the location of the smartphone THEN that application may have malicious code. The rule generated for this case is shown below:

IF Not ($SEND\_SMS$) && ($ACCESS\_FINE\_LOCATION$) THEN Malware

Here, malicious code and malware are interchangeable terms. The possible outcomes are: Goodware or Malware. Nevertheless, the proposed infrastructure might be capable of evaluating a third option, Adware, in a few cases. In this paper we do not describe the IF THEN rules for the third kind of outcome. In this work, we restrain the possible outcomes to the two mentioned options.

We have used 7 rules in our experimentation which are listed in Table 2.1 (note that rules 1 & 2 are the same). We have listed in Table 2.1 the most frequently used rules. They mainly cover cases of user information leakage.

The most frequently used detection rules that we have utilized in our experimentation are listed in Table 2.1 (second column).

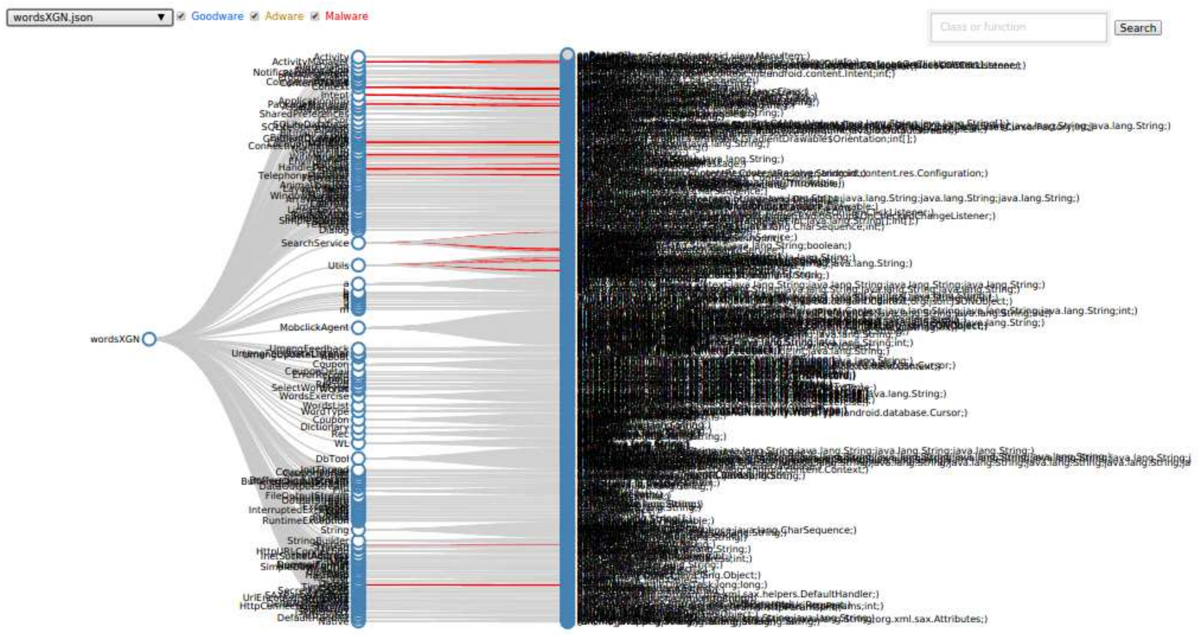


Figure 2.10: Visualization of the *DroidKungFu1* malware with full features chosen (i.e., all the checkboxes are activated).

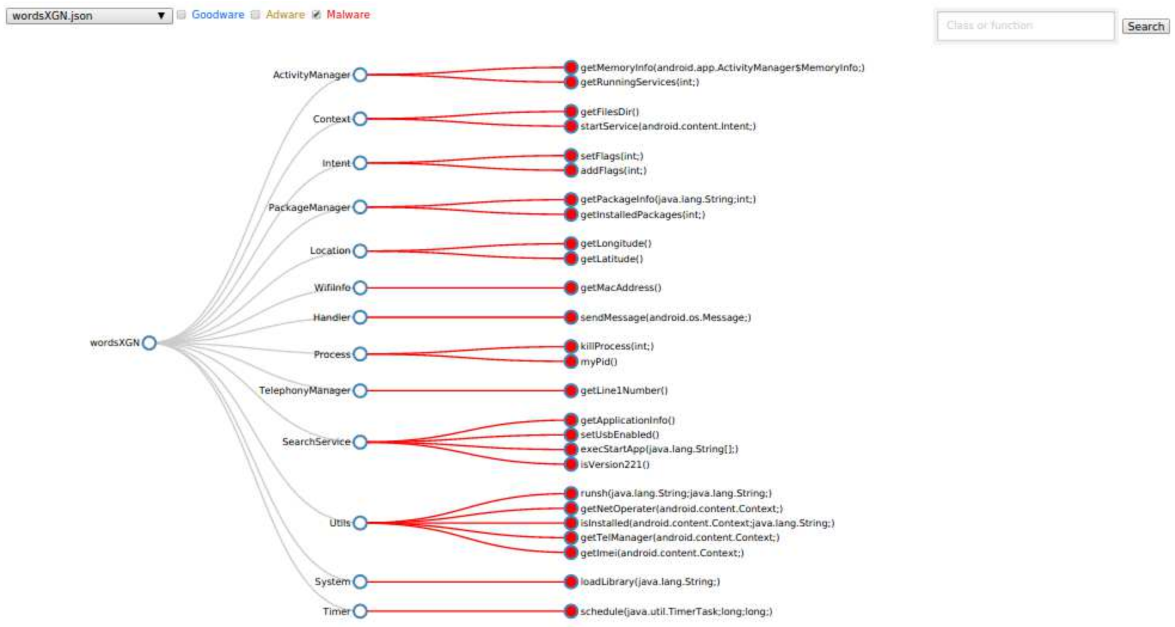
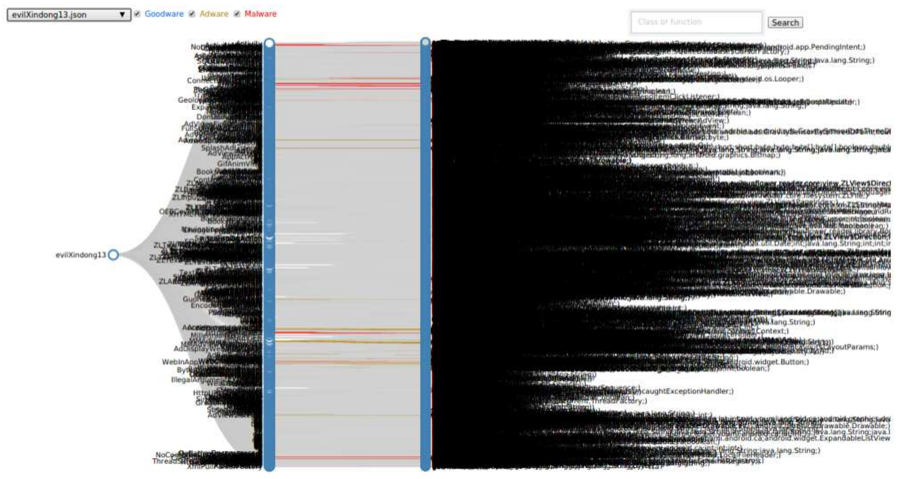

wordsXGN.json ▼
☐ Goodware
☐ Adware
☑ Malware
Class or function
Search
wordsXGN
ActivityManager
getMemoryInfo(android.app.ActivityManager$MemoryInfo;)
getRunningServices(int;)
Context
getFilesDir()
startService(android.content.Intent;)
Intent
setFlags(int;)
addFlags(int;)
PackageManager
getPackageInfo(java.lang.String;int;)
getInstalledPackages(int;)
Location
getLongitude()
getLatitude()
WifiInfo
getMacAddress()
Handler
sendMessage(android.os.Message;)
Process
killProcess(int;)
myPid()
TelephonyManager
getLine1Number()
SearchService
getApplicationInfo()
setUsbEnabled()
execStartApp(java.lang.String[];)
isVersion221()
Utils
runsh(java.lang.String;java.lang.String;)
getNetOperater(android.content.Context;)
isInstalled(android.content.Context;java.lang.String;)
getTelManager(android.content.Context;)
getImei(android.content.Context;)
System
loadLibrary(java.lang.String;)
Timer
schedule(java.util.TimerTask;long;long;)

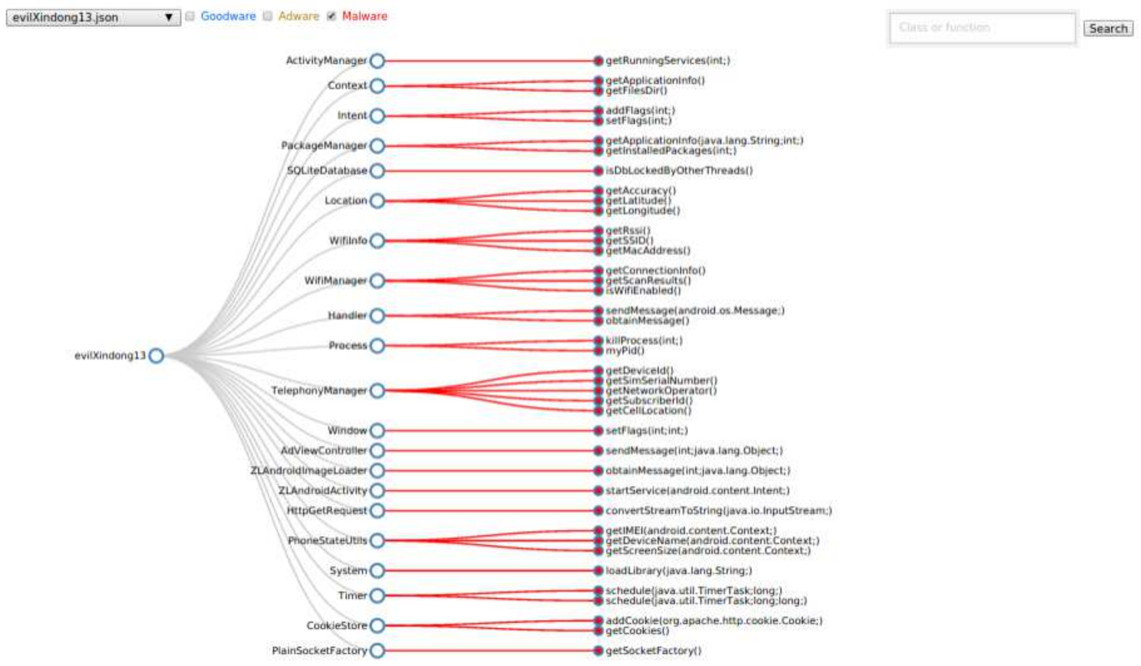
Figure 2.11: Visualization of the malicious API calls detected by our system for *DroidKungFu1*. Note the chosen options of the monitored malware in the dendrogram at the upper left side. First, we shorten version of the package name (*wordsXGN*) of the malware in the combobox. Next we have three checkboxes, namely: Goodware, Adware, and Malware. In this graph, only the red checkbox has been activated in order to conduct the visual analysis. The full package name of *DroidKungFu1* is *com.nineiworks.wordsXGN*.

2.5.4 Client-side CPU use analysis

We define the CPU utilization of a given application as the ratio between the time while the processor was in busy mode only for this given application both at the user and kernel levels, and the time while the processor was either in busy or idle mode. The CPU times have been taken from the Linux kernel through the files "/proc/stat" and "/proc/pid/stat" where pid is the process id of the given application. We have chosen to sample the CPU utilization every second.



(a) Dendrogram in full features (Goodware, Adware and Malware) for *DroidKungFu4*.



(b) Graph visualization of the detected malware in the case of *DroidKungFu4*.

Figure 2.12: Dendrograms of the tested application. (a) Upper diagram: Graph of the *DroidKungFu4* in full features, and (b) Lower diagram: Graph of the malicious functions invoked by *DroidKungFu4*.

The CPU utilization of the Sink application has been measured in order to evaluate the cost of receiving the partial traces from the diverse monitored applications, processing them and persisting them in the SQLite database varying the time interval between two consecutive partial traces sent. We expect to see the CPU utilization of the Sink increase as the time interval between two consecutive partial traces sent decreases. Indeed, since the Sink must process more partial traces, it needs more CPU resource. This is confirmed by the curve in Figure 2.13. The CPU utilization has a tendency towards 30% when the time interval between two consecutive partial traces received tends to 10 ms because the synthetic application takes almost 30% of the CPU for building and sending partial traces, and the rest of applications utilize the rest of the CPU resource. When no monitored applications send partial traces to the Sink and the Sink is running in the background (i.e., its activity is not displayed on the screen), it consumes about 0% of the CPU.

The CPU utilization of a synthetic application has also been measured in order to evaluate the cost of building and sending the partial traces to the Sink while the time interval between two consecutive partial traces sent was varied. We expect to see a higher CPU utilization when the application is monitored. Indeed, since the synthetic application must build and send more partial traces, it needs more CPU resource. This is confirmed by the Figure 2.14. We note that the increase of CPU utilization of the application can be up to 28% when it is monitored. The chart shows an increase in application CPU utilisation to a level up to 38% which is justified when the monitoring is fine-grained at 10 ms. However, this high frequency is not likely to be needed in real applications.

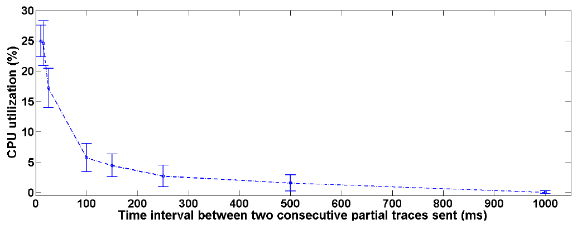


Figure 2.13: CPU utilization of the Sink.

2.5.5 Responsiveness

We define an application as responsive if its response time to an event is short enough. An event can be a button pushed by a user. In other words, the application

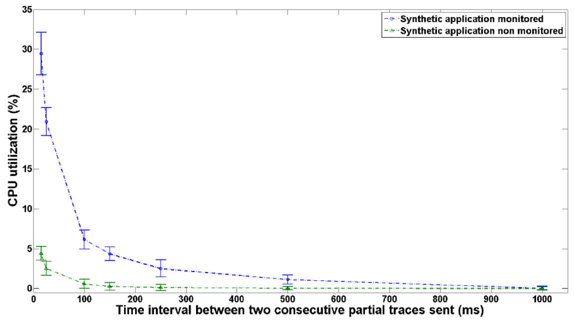


Figure 2.14: Difference of CPU utilization between an application monitored and non monitored.

is responsive if the user does not notice any latency while the application is running. In order to quantify the responsiveness and see the impact of the monitoring on the responsiveness of monitored applications, we have measured the time spent for executing the prologue function of the synthetic application. We have evaluated the responsiveness of the monitored application when the Sink was saturated by partial traces requests, i.e. in its worst case. The measured response time was in average less than 1 ms. so, the user does not notice any differences when the application is monitored or not, even though the Sink application is saturated by partial traces. This is explained by the fact that UDP is connectionless and therefore sends the partial traces directly to the UDP socket of the Sink without waiting for any acknowledgments.

2.6 Limitations

So far we illustrated the possibilities of our visual analysis framework by analyzing 8 existing malicious applications. We successfully identified different types of malware accordingly to the malicious payload (e.g., privilege escalation, financial charges and personal information stealing) of the App while using only dynamic inspection in order to obtain the outcomes. Even though the results are promising, they only represent a few of the massive malware attacking today's smart devices. Of course, the aim of this system is not to replace existing automated Android malware classification systems because the final decision is done by a security analyst.

Although, here, we propose a malware detector system based on runtime behaviour, this does not have detection capabilities to monitor an application's execution in real time; so this platform cannot detect intrusions while running. It only enable detecting past attacks.

Also, one can figure out that malware authors could try avoiding detection, since they can gain knowledge whether their App has been tampered with or no. As a result, the actual attack might not be deployed, which may be considered a preventive technique. Moreover, it is possible for a malicious application to evade detection by dynamically loading and executing dalvik bytecode at runtime.

One of the drawbacks of this work could be the manual interactions with the monitored application during runtime (over some time interval). Also, the classification needs a more general procedure to get the rule-based expert system. The natural next step is to automate these parts of the process. For example, in literature there are several approaches that can be implemented in order to automatically generate more IF-THEN rules [41] or to resort to the MonkeyRunner kit available in Android SDK to simulate the user interactions. Of course, the outcomes of the 8 sample malware presented here are limited to longest time interval used in the study, which was 10 minutes. Extending this "playing" time with the App using tools for the automation of user's interactions could provide a more realistic graph and better pinpoint the attacks of the mobile malware.

Another limitation of this work is that it can only intercept java level calls and not low level functions that can be stored as libraries in the applications. Thus, a malicious App can invoke native code through Java Native Interface (JNI), to deploy attacks to the Android ecosystem.

It is worth mentioning that our API hooking process does not consider the Intents. The current version of the infrastructure presented in this paper is not capable of monitoring the Intents sent by the application, as sending Intents does not require any kind of permission. Not being able to monitor Intents means that the infrastructure is not able to track if the monitored application starts another app for a short period of time to perform a given task, for instance opening a web browser to display the end-user license agreement (EULA). Also, adding this feature would allow knowing how the target application communicates with the rest of the third party and system applications installed on the device. In [76], the authors discuss the effectiveness of the analysis of Android Intent in malware detection.

Ultimately, this framework could be useful for final users interested in what Apps are doing in their devices.

2.7 Conclusions of the Chapter

We provide a monitoring architecture aiming at identifying harmful Android applications without modifying the Android firmware. It provides a visualization graph named Dendrograms where function calls corresponding to predefined malware behaviors are highlighted. Composed of four components namely, the embedded client, the Sink, the Web Service, and the visualization, any Android application can be monitored without rooting the phone or changing its firmware.

The developed infrastructure is capable of monitoring simultaneously several applications on various devices and collecting all the traces in the same place. The tests performed in this work show that applications can be prepared to be monitored in a matter of minutes and that the modified applications behave as they were originally intended to, with minimal interference with the permissions used for. Furthermore, we have shown that the infrastructure can be used to detect malicious behaviors by applications, such as the monitored *FakePlayer*, *DroidKungFu1* and *DroidKungFu4* and the *SMSReplicator* and many others taken from the dataset of the Android Malware Genome project.

In the visualization part we have opted for the use of dendrogram diagrams because it is easier, so far, to analyze than the graphs in Neo4j.

Evaluations of the Sink have revealed that our monitoring system is quite reactive, does not lose any partial traces, and has a very small impact on the performance of the monitored applications.

A major benefit of the approach is that the system is designed as platform-independent so that smart devices with different versions of Android OS can use it. Further improvements on the visualization quality and the user interface are possible, but the proof of concept implementation is demonstrated to be promising.

Chapter 3

Dynamic DNS request monitoring of Android malware

Of course, the first task in detecting network attacks and instructions is to collect security-related data. This chapter proposes two methods of capturing the DNS requests done by the smart devices to a remote machine, this is crucial in order to monitoring the Android malware. Most of the Android-based malware communicate with some remote servers (command and control center) either for getting instructions as it does in a botnet or to send data / information stolen from device to the attacker. As stated in [77]: "Whatever intention is there, this malware are mostly dependent on the remote machine and always need to communicate with it." The first method is based on the development of an Android sniffer for DNS Request Monitoring of Android using network traffic. As it is defined in [78]: "A packet analyzer (also known as a packet sniffer) is a computer program or piece of computer hardware that can intercept and log traffic that passes over a digital network or part of a network". To the best of our knowledge, this first non-rooted sniffers that captures and intercepts URLs requested by the smart devices to remote machines, publicly available for the Android ecosystem. Later, we also develop a second method based on the hooking of API calls. So, in the latter case the objective we pursue is to try to discover which API calls involve a query to a DNS server (using the InetAddress class), to capture the URL to which the malware wants to perform some action (e.g., stealing, update, data transmission or leakage, etc.). With these both approaches, we could perform a pattern matching with the logs of DNS servers of a mobile operator, and we could detect the same behavior in other users without the need to install the monitored App into theirs smart devices in most of the time. The tests that we have carried out help to identify the URLs

(Uniform Resource Locators) invoked through the DNS queries requested by the smartphone malware.

3.1 Introduction

As aforementioned, in the last years, mobile smartphones with a mobile OS are widely being used, and now we observe the explosive growth of mobile devices around the world. Accordingly, to Symantec's Internet Security Threat Report(2018), Internet security threats such as mobile malwares are rapidly increasing and diversifying as well.

On the one hand, to conduct our experiments with real malware samples, we use the MalGenome dataset in order to generate network traffic, among other issues. Besides, most of the malware we examine misuse the DNS in order to obtain the URLs of their command and control servers. Then, the problem of determining the DNS queries done by the malware through devices without modifying the firmware or rooting smartphone is very important and it poses a big challenge. From traces we generated from Apps under test, we can extract malicious URLs invoked by the malware. On the other hand, most of the research published in the technical literature focuses on host-based malware detection systems, which implies an in-depth analysis of malicious applications; instead of also integrating in the analysis of the system, the traffic or behavior of the Apps in the communications networks; which severely disables the detection of malicious activities that occur on the mobile device through the Internet. In this is remarkable, the abusive use by the mobile malware of the Domain Name System (DNS), one of the fundamental components of the Internet.

The main objective of this chapter is to propose two methods, on the side of the smartphone, that allows the combination and the correlation of two complementary approaches: the top-bottom detection by identifying the names of malware domains through the records of the DNS servers, and bottom-up detection using classic dynamic analysis in Android applications to identify malware. Herein, we address the bottom-top approach. Concretely, in monitoring malware, specialized tools are needed to extract information from network traffic of the intelligent mobile device without modifying its firmware or "rooting" it. Specifically, in this chapter we focus on the detection of "bottom-up" malware, using dynamic analysis and capturing DNS queries carried out by smartphones with malicious remote servers, i.e., Dynamic DNS request Monitoring of Apps.

3.1.1 Problem definition

The problem to solve is how to capture network traffic from the smartphone. In our case, we are interested in identifying the URLs invoked through the DNS queries required by the malware, without altering the firmware or "rooting" the intelligent device. On the other hand, it is commonly believed that the next generation of mobile malware detection systems will combine malware analysis, anomaly detection, network analysis, log analysis, as well as detection of denial of distributed services (DDoS), among other aspects; all in order to allow real-time monitoring of malware attacks. Here, we focus on Dynamic DNS request monitoring of Android malware by capturing traces from the smartphones.

So, at present, there is the challenge of developing a scalable and high efficient platform for Android monitoring and analysis of security events that can compromise mobile devices or pose threats that affect the operator's infrastructure. The management of the detection and reaction to new threats and their mass spread due to the ubiquity of the underlying communications network, will be done by extending security event management capabilities for the monitoring, detection, characterization, and mitigation of threats to mobile devices, as well as creating an early warning system for operators. In particular, this research will address part of this challenge, showing how to obtain the traces of network traffic from mobile device Apps under the restrictions mentioned above.

3.1.2 Contribution and Outline of the chapter

Throughout this chapters we present two novel methods for detecting malware, by capturing malicious URLs at the network level of intelligent devices by executing the functions they call, and developing an ad hoc Android sniffer. The first contribution [58], it is the method description and the proposed malware detection tool (Android sniffer) for Dynamic DNS request Monitoring of Android Applications via networking without rooting or modified the phone under trial. The second contribution of this chapter is the enhancement of the Android platforms described in [60], consisting of an implementation on the side of the smartphone and on the side of the remote server. Here, we concentrate on capturing the URLs requested to detect malicious transactions initiated by an application running on the Android phone side. In particular, we conduct the processing on the smart device side at the network of the Apps in order to do the dynamic DNS request monitoring of Android malware via instrumentation.

The rest of the chapter is organized as follows: Section II discusses the related works. Section III presents the design of the first method based on a VPN-platform to capture the network packets. Thereafter, in Section IV, we have the second method based on the instrumentation of native functions of Android OS. Section V evaluates the performance of both methods and presents a comparison of them in terms of effectiveness and efficiency. The battery consumption and the use of some smartphone resources are also considered. Finally, Section VI presents the remarking conclusions of this chapter.

3.2 Related Work

In this section, we introduce some previous works on this topic. The analysis and detection of Android malware have been a hot theme of research in the last years. Several concepts and techniques have been proposed to counter the growing amount and sophistication of this malware.

There are several ways to dynamically intercepting/obtaining the network packets transmitted by the mobile smart devices to remote servers, among others, namely: i) by using a proxy [80], ii) by utilizing an Android network log monitor [13], or packet analyzers (sniffer) such as *tPacketCapture Pro* [81] & *MalDetec* [82] based on Virtual Private Networks (VPN) approaches, iii) by modification and customization of the Android OS as in [77], iv) by hooking function calls, such as library APIs, (e.g., OpenConnection method), and v) by exploiting the Logcat tool from Android OS such as the tool dubbed *Logdog*, where Logcat is the command to view and filter information from the Android logging system [83].

In [77], Rughani customized source code of Android OS [1]. The customization includes modifying the code in needed files and rebuilding the code to make custom OS. Afterward, the author utilizes a python script that captures and intercepts logs. It then extracts IP Address / URLs from the logs and puts them in a file. After extracting information, it compares the extracted information with existing blacklisted IP Address (which are downloaded from openbl.org automatically by the script). As the last step, the script creates result file containing suspicious IP Addresses (if any found). Collected information is not restricted. Unlike the work in [77], with the "approaches" presented in this chapter later (See Sections 3.3 and 3.4) we can obtain the URLs consulted by the smartphone, without modifying the firmware of the Android OS making use of dynamic detection techniques

[1] Available at AOSP (https://source.android.com/)

using instrumentation (introducing hooks) in API function calls and non-rooted Android sniffer.

In the approach proposed by Bae et al. [13], they minimize the use of high overhead functions and replace them to lightweight features (e.g., function call monitoring). They have leveraged those features instead of using high overhead operations. Also, they monitor the network connections, including the DNS queries requested by using an Android network log monitor. Our approach is different from theirs, in that they [13] need to root the smartphones while we can utilize our sniffer and dynamically obtain the URLs requested by the Android phone, without rooting the smart devices as it is described in Section 3.3.

Let us begin with the methods of interest for dynamic DNS request monitoring of Apps. First, the method hooking is a technique used to intercept the call of a certain method at runtime to change the behavior of the calling application. By dynamically (it is used for mechanisms that can dynamically apply a hook at runtime) intercepting function calls frameworks can analyze both single calls and sequences of calls to reconstruct behaviors for semantic representations or monitor the function calls for misuse. Function hooks can also be used to trigger additional analyses. For instance, if a function was hooked and triggered, parameter analysis could then be applied to retrieve the parameter values of when the function was invoked. The analysis framework InDroid [84] inserted function call stubs at the start of each opcode's interpretation code in order to monitor bytecode execution and analyze Android behaviors. While it does require modifications to the Dalvik VM and may not work on Android 5.0 (e.g., with ART), the method requires relatively light modifications and has been used on versions 4.0-4.2. However, InDroid requires to root the smartphone, which it is not necessary in our case. Dynamic hooking happens in volatile memory only.

On the other hand, of course, we could try to install a limited version of Wireshark [85] for Android, which allows us to identify the URLs invoked through the DNS queries requested from the Smartphone. However, this would imply "rooting" the phone, which is not acceptable for practical purposes in our case, since mobile operators could not maintain the guarantees of their users. We have also done tests with *tPacketCapture Pro* [81], very similar to *tcpdump* [86], that allows the capture of Internet traffic back and forth on the smartphone, but it has severed restrictions to show the DNS queries made by the Android phone. Also *tcpdump* requires "rooting" the smart mobile device. Another possibility found in the review of the technical literature is *Logdog* [83], which has been developed to detect botnets for smartphones using log analysis techniques. Again, *Logdog* is based on a collection of Android logs called *Logcat* and you need to get the permission of superuser (root user) of the OS, to be able to operate with this App on the smart device, which is out of scope of this thesis work.

Furthermore, in [87], Brandolini designed and implemented a security library for Android applications exploiting the hooking of Java and native functions to enable runtime analysis. The library verifies if the application shows compliance to some of the most important security protocols, and it tries to detect unwanted activities based on the Dalvik compiler. Testing of the library shows that it successfully intercepts the targeted functions, thus allowing to block the application malicious behavior. He also assesses the feasibility of an automatic tool that uses reverse engineering to decompile the application, inject his library, and recompile the security-enhanced App.

Moreover, as aforementioned Android 5.0 introduced the new ahead- of-time compiling Android runtime ART. So, it is needed an advanced instrumentation approaches based on the new virtual machine ART which are addressed in [88, 89, 90, 91].

In the forthcoming two subsections, our aim is the detection of dangerous DNS queries requests in Android smart devices.

3.3 Method1: Dynamic DNS request monitoring of Android malware via networking

As stated before, currently, almost anyone has access to smart devices such as smartphones, since these devices are used for all kinds of daily activities, ranging from social networks to banking and business transactions. At the same time, unfortunately, smart devices are a direct target for malicious activities of hackers and cybercriminals that aim to distribute malware to the typical user, who does not know the intentions behind it. Since these attacks usually install on your device malware from third-party (e.g., unofficial) reservoirs, typically free of charge, thus causing monetary losses such as financial fraud, theft of user profiles and corporate or personal data, user spoofing, adware and ransomware attacks, among others. All the above affectations without the user realizing it, sending the stolen information through the network where the smartphone is connected.

One of the most common mechanisms used by malware is that once installed on the device, connect to the network and send a request signal to a remote online server. This master machine gives instructions on how to proceed to attack the smart mobile device, either by capturing network traffic or by installing other Apps without the user's consent; often leaving the smart device unusable or obtaining confidential information. These Apps do not usually show suspicious activities so they can be tricks by running naturally while executing their second intention; the user then becomes totally vulnerable to attackers who can take some control of their device by simply accessing the network.

3.3.1 Network Sentinel: The Proposed Malware Detection Tool via networking

Although there are proposals that aim to control network traffic in smart devices [69, 92], current smartphones do not usually give importance to network requests and what they could represent to them in terms of mobile security. Also, ordinary users must resort to sophisticated systems and maintenance techniques to ensure that their smart devices are not infected with some malware. Therefore, it is of great value and importance to develop an application that can track and monitor network traffic in a more automated way and that performs the majority of actions necessary to give the state of reliability of the equipment to the user. In this way we can benefit both common users and some more advanced users who need specific administrative tools in their smart devices, some as developers or researchers can make use of the application to help them understand how packets travel through the network.

So, the main goal here is, develop an App that manages to keep track of network packets or network traces in order to ensure the degree of reliability of the applications installed on the smart device and avoid or warn possible malware problems associated with communication with malicious servers in the network. We have developed an App that can track access to the network that other Apps can make, whether the user has proof of these accesses or not. The Proposed Malware Detection Tool captures DNS queries requested by Apps on the smartphone to remote online servers, and it creates a check that defines to the user whether or not there is some unwanted access by the installed applications. *Network Sentinel* is an application freely available for Android mobile phones. While running *Network Sentinel* allows the user to capture Internet traffic sent to and from the smartphone similar *tcpdump* and save this to a PCAP file. A conceptual block diagram of the proposed tool for malware detection via networking is shown in Figures 3.1.

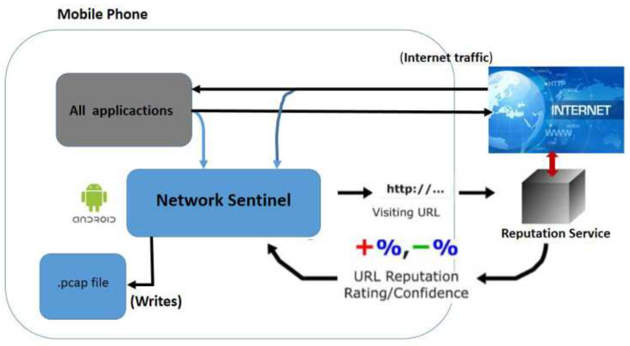


Figure 3.1: Usage of the proposed tool termed *Network Sentinel*.

Let us describe the basics of the proposed method, where we develop a simple sniffer based on the Android Studio IDE version 2.3. Our application stars gathering the network traffic of the targeted App or Apps at the smartphone, through the creation of a local virtual private network (VPN), See Figures 3.2 and 3.3. We then need to use *VpnService* to redirect all the device's network traffic through our application. Of course, we can only capture the DNS traffic from/to the App or Apps under test. Afterward, the network traffic captured and all requests that exist within the device are thus packaged it into a PCAP file (PCAP is a generic API for capturing network traffic). After that, the *jNetPcap* library is utilized; which is an open source Java library, used to capture and decode network packets. And, it uses native implementations to provide optimum packet decoding performance. In Figure 3.2, we have a diagram showing how the *VpnService* is called from *Main Activity* of the Android programming and its services that this utilized.

By doing so, the obtained IP traffic or URLs in the network traces can be checked through a blacklist server, and know if there is any malicious behavior on the device. Consequently, it is possible to define/declare which application under test is not suitable to use or uninstall. Note that *Network Sentinel* runs in its entirety on the local smart device and traffic is not routed through a remote VPN server. This procedure is the basics of the proposed malware detection tool for Dynamic DNS request monitoring that works with the ART compiler. And, we named as to

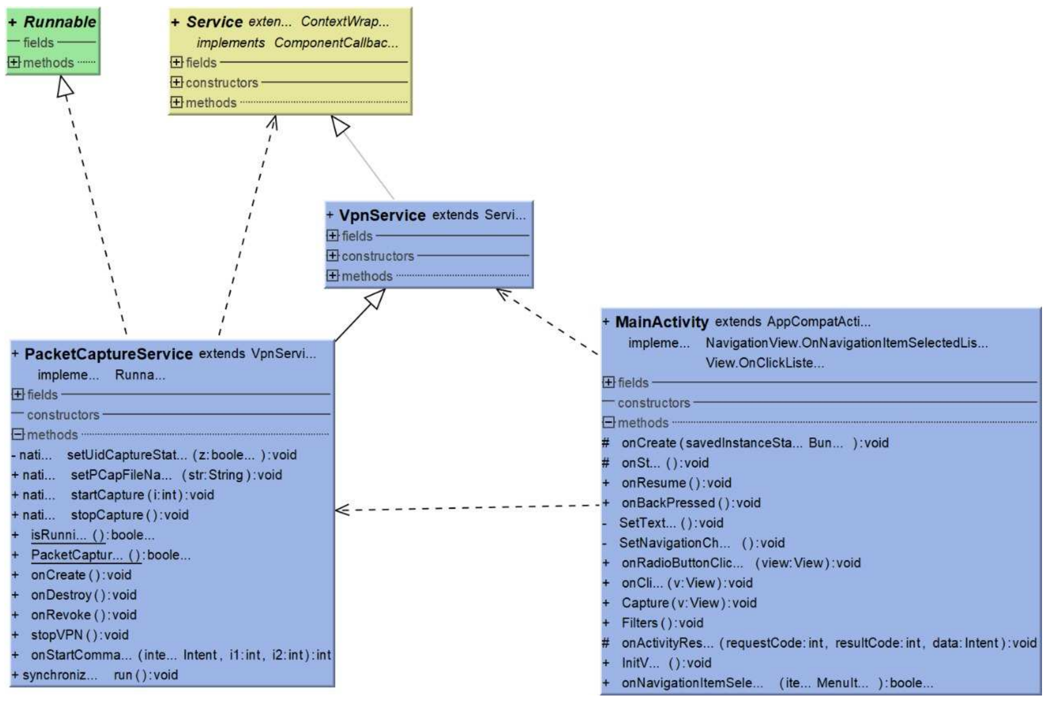

+ Runnable
fields
methods
+ Service exten... ContextWrap...
implements ComponentCallbac...
fields
constructors
methods
+ VpnService extends Servi...
fields
constructors
methods
+ PacketCaptureService extends VpnServi...
impleme... Runna...
fields
constructors
methods
- nati... setUidCaptureStat... (z:boole...):void
+ nati... setPCapFileNa... (str:String):void
+ nati... startCapture (i:int):void
+ nati... stopCapture ():void
+ isRunni... ():boole...
+ PacketCaptur... ():boole...
+ onCreate ():void
+ onDestroy ():void
+ onRevoke ():void
+ stopVPN ():void
+ onStartComma... (inte... Intent, i1:int, i2:int):int
+ synchroniz... run ():void
+ MainActivity extends AppCompatActi...
impleme... NavigationView.OnNavigationItemSelectedLis...
View.OnClickListe...
fields
constructors
methods
# onCreate (savedInstanceSta... Bun...):void
# onSt... ():void
+ onResume ():void
+ onBackPressed ():void
- SetText... ():void
- SetNavigationCh... ():void
+ onRadioButtonClic... (view:View):void
+ onCli... (v:View):void
+ Capture (v:View):void
+ Filters ():void
# onActivityRes... (requestCode:int, resultCode:int, data:Intent):void
+ InitV... ():void
+ onNavigationItemSele... (ite... MenuIt...):boole...

Figure 3.2: The diagram of use and implementation of the capture service of the VPN at the smart device.

Network Sentinel which needs at least the Android version number 5.0 (code name Lollipop) to work correctly.

Of course, VPN approaches [81, 92] have suggested in the literature to provide secured communications in Android ecosystems. For instance, *PrivacyGuard* [92], an open-source VPN-based platform for intercepting the network traffic of Apps. This Android application also requires neither root permissions nor any knowledge about VPN technology from its users. Thus, we implemented *Network Sentinel* on the Android platform by also taking advantage of the *VpnService* class provided by the Android SDK. *PrivacyGuard* is an App that alerts you when one of our Apps leaks sensitive information to a remote server, which is a little bit different from our scope of the proposed method presented here. Furthermore, we tested the paid App *tPacketCapture Pro* [81], but we did not succeed to get or capture straightforward any URLs at all. So, this was one of the reasons to develop *Network Sentinel* to fulfill this gap in the arsenal of the tools publically available for the Android analysis, in particular for Dynamic DNS request monitoring of Android malware. In Figure 3.4, it can be seen the icon on the *Network Sentinel* as is depicted in the platform of Google play store. This Android "sniffer" is available

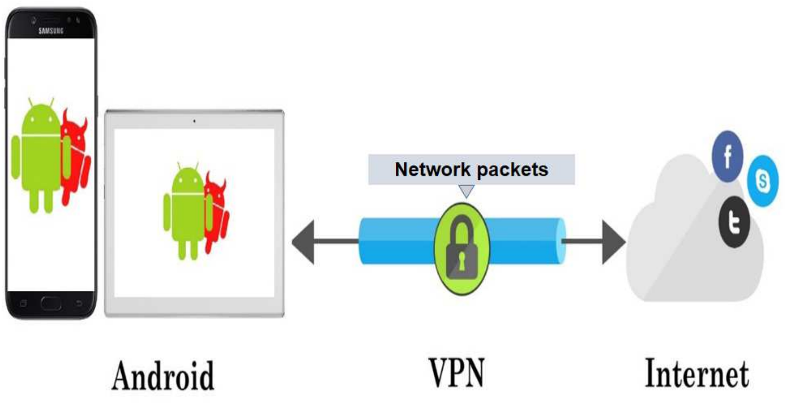


Figure 3.3: Schema handling of packets via VPN (Virtual Private Network) from Android smartphone or Tablet. The main configuration of our Android sniffer is shown, capturing packets through a protected data tunnel.

through Google Play Store, at the web (december 2018) link [2].

3.3.2 Experimental Results

In order to evaluate our approach (Android sniffer), we conduct several experiments with the two smartphones, XiaomiRedmi 3S Prime and Samsung Galaxy Grand Prime. First of all, we validate the network traces (benign Apps and malware) provided by our tool. Second, we carry out interceptions of URLs by applying Dynamic DNS request monitoring of the App under test combined with existing blacklisted URL/IP address of malicious Domain Name available on the Internet, to detect the presence of malware. And third, we address the consumption of resources of the Android "sniffer", mainly through the power consumption of the device regarding the battery usage.

3.3.2.1 The capture of the Requested URLs

We first present the main menu of the *Network Sentinel* as can be shown in Figure 3.5. Here we start by choosing the configuration of the dynamic monitoring, e.g. we can select the App under examination (we can also monitor several Apps, if we want to do that) and the type of capture of the network traces.

[2]https://play.google.com/store/search?q=Network%20Sentinel&c=appstext

Figure 3.4: The Android sniffer termed *Network Sentinel* is available at Google Play Store.

So far, the menu of *Network Sentinel* is only available in the Spanish language, however, future versions are going to migrate to a full version in the English language. For instance, in this case, "**INICIAR CAPTURA**" translated into English is "Start Capture", and "**SELECCIONAR APLICACIONES**" means in English "Choose the App or Apps to monitor". It is important to mention that the *Network Sentinel* provides information about the protocols in use in real-time, not the raw data. After obtaining the network traces, the results are being automatically saved in a file on the smartphone (Android sdcard) with an extension PCAP. We validate *Network Sentinel* by comparison, its results against similar monitoring using an Android version of the protocol analyzer Wireshark [85] in an ad hoc set up with the smartphone Samsung Grand Prime rooted. Of course, doing the validation with *tcpdump* is convenient as well, because we have the Android tcpdump available, which is a command line packet capture utility but it requires root privileges. We tested 10 benign Apps (they were taken from Google Play Store) and 10 malicious software.

3.3.2.2 The Maliciousness of the Android application

Figure 3.6 depicted the functional block diagram employed the feature of inspecting the threat of the App under examination in *Network Sentinel*, utilizing blacklisted service provided by the Internet. We can further extend the malware analysis selecting the option "Usar Servidor DNS" in the configuration menu of *Network Sentinel*, here we can either choose one of two possibilities namely, Web of Trust (WoT) [93] or Safe Browsing [94]. In Figure 3.7, it is shown the monitoring results

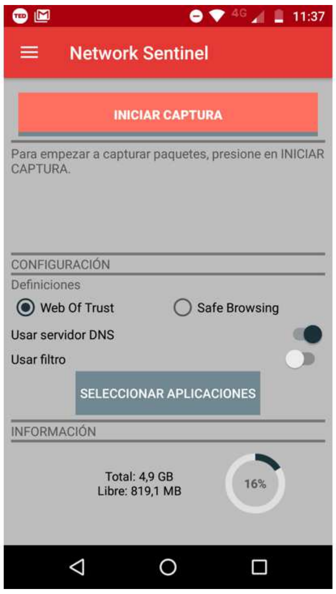


Figure 3.5: The main menu of *Network Sentinel* available at Google Play Store.

using the WoT, we reach this feature in *Network Sentinel* by using the *PCAP Analisis*, in particular with the option *THREAT*. Usually, a web site in red color could imply a suspicious or malicious URL.

3.3.2.3 Battery Usage with *Network Sentinel* working

Application performance is always a trouble for everyone developing Apps. Here we review the battery usage of the *Network Sentinel* by utilizing the tools *Batterystats* and *Battery Historian*. *Batterystats* takes data from our smart device bout battery. On the other hand, *Battery Historian* converts this data to HTML format to be able to see it on Browser. *Batterystats* is a part of the Android framework and *Battery Historian* is on Github as opensource at https://github.com/google/battery-historian.

The step by step procedure how we use *Batterystats* is described at the following web site below[3]. During the experiments using *Batterystats* & *Battery Historian* we "play" with the App under test for about 15-20 minutes. The typical outcome of the battery usage during the trials was roughly an average of 0.05%, as it is shown in the Table 3.1, an excerpt of the *Battery Historian* tool.

[3]https://medium.com/@elifbon/android-application-performance-step-4-battery-b1f88d096b1e.

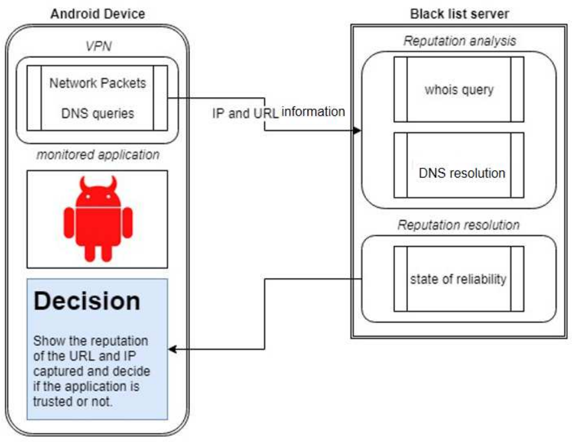


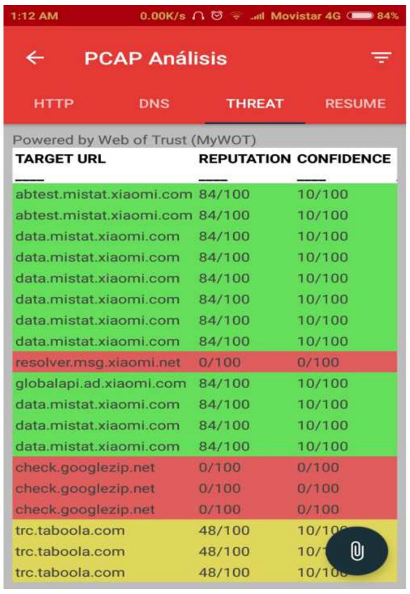
Figure 3.6: Functional diagram of the developed App. Configuration of the structure of the App and connection to the blacklisted service.

3.4 Method2: Dynamic DNS request monitoring of Android malware via instrumentation

The analysis of Android applications becomes more and more difficult currently. Both benign and malicious developers use various protection techniques, such as Java reflection, dynamic code loading, and code obfuscation, to prevent their Apps from reverse engineering. Besides, in order to build a hooking framework for dynamic analysis, we follow a similar hooking approach as the propose by Brandolini in [87]. Throughout this section, we present a method for detecting malware, by capturing malicious URLs at the level of intelligent devices by executing the functions they call. The contribution of this method2 is to expand the system for Android platforms described in [60], consisting of an implementation on the side of the smartphone. Here, we concentrate on capturing the URL queries requested to detect malicious transactions initiated by an application running on the Android phone side.

3.4.1 AppURL: The Proposed Malware Detection Tool via instrumentation

Next, the methodology used in this section is described. This consists of three phases. The first phase is the identification and capture of URLs invoked through the DNS queries requested that are being made by an Android application under


1:12 AM 0.00K/s Movistar 4G 84%

PCAP Análisis

HTTP DNS THREAT RESUME

Powered by Web of Trust (MyWOT)

TARGET URL	REPUTATION	CONFIDENCE
abtest.mistat.xiaomi.com	84/100	10/100
abtest.mistat.xiaomi.com	84/100	10/100
data.mistat.xiaomi.com	84/100	10/100
data.mistat.xiaomi.com	84/100	10/100
data.mistat.xiaomi.com	84/100	10/100
data.mistat.xiaomi.com	84/100	10/100
data.mistat.xiaomi.com	84/100	10/100
data.mistat.xiaomi.com	84/100	10/100
resolver.msg.xiaomi.net	0/100	0/100
globalapi.ad.xiaomi.com	84/100	10/100
data.mistat.xiaomi.com	84/100	10/100
data.mistat.xiaomi.com	84/100	10/100
data.mistat.xiaomi.com	84/100	10/100
check.googlezip.net	0/100	0/100
check.googlezip.net	0/100	0/100
check.googlezip.net	0/100	0/100
trc.taboola.com	48/100	10/10
trc.taboola.com	48/100	10/
trc.taboola.com	48/100	10/10

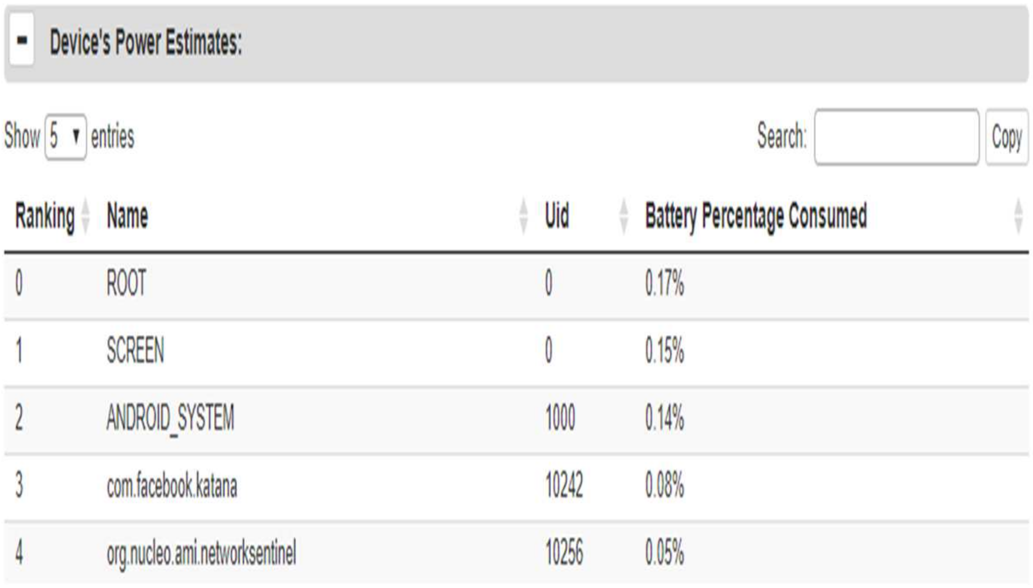
Figure 3.7: Outcome of the queries to the blacklisted service WoT [93] with *Network Sentinel*.

test from the smartphone. The second phase is to determine if the URLs obtained in the first phase are whether benign or malicious, by using a reputation grading (blacklisted) service available at the Internet. Finally, the third phase consists of storing the malicious URLs, in a text file and in a database for later use, if applicable. That is, if any malware detection is made.

3.4.1.1 First Phase: Acquisition of data, Analysis and Monitoring of mobile traffic at the network level

In this phase, it has been designed and developed a module to capture data from mobile applications (in our case we will conduct our experiments by using the popular MalGenome dataset [36]), which serves to identify attacks against mobile platforms. In order to make the system as non-intrusive as possible, first, we obtain a general map about the nature of the applications installed on the client's device. This map is obtained with a simple query to the Virustotal API [96]. On the other hand, there are applications whose analysis gives an unknown result in *VirusTotal*. In this case, method2 would be executed to monitor the suspicious App, performing an instrumentation process, based on the injection of functions that cause the sending of records of its execution to a central server. This last process is carried out through a process called "Hooking". Unlike *method1* described in 3.3, in our case (*method2*) the hooking process is still done partially by manual

Table 3.1: Table of the battery usage with the smart device Xiaomi Redmi 3S, in particular the **org.nucleo.ami.networksentinel** (Ranking 4 in the Table has as **Battery Percentage Consumed**, 0.05%).



- Device's Power Estimates:

Show 5 entries Search: Copy

Ranking	Name	Uid	Battery Percentage Consumed
0	ROOT	0	0.17%
1	SCREEN	0	0.15%
2	ANDROID_SYSTEM	1000	0.14%
3	com.facebook.katana	10242	0.08%
4	org.nucleo.ami.networksentinel	10256	0.05%

sequence, so far App by App, which an inconvenient. Here, we test hooking Java methods and native functions to enhance Android applications security.

In our case, the instrumentation process is conducted by using ADBI toolkit [95], which implements the hijacking utility of the ARM binary code [97]. Due to its binary-level hooking feature, the ADBI can hook the native code of an App, which is required in order to instrument the URL queries requested. Here, we need to invoke native functions as it is explained as follows. A detailed explanation of the monitoring architecture used is available in [60]. The development of this system allows to analyze in a visual way the behavior and the dangerous functions executed by the application under study. Next, the task will be to identify and evaluate relevant sources of information security in the mobile network, and show how they could be used to detect security events related to subscribers in the mobile network. Considering the current traffic rates, the detailed inspection of user plane traffic is prohibitive, in addition to that the privacy implications and regulatory limitations have to be enforced or followed. Thus, the data source considered here is that of the DNS queries made by the App under monitoring on the smartphone. In general, DNS records contain very valuable information about domain names and associated IPs that Apps consult. It will therefore contain

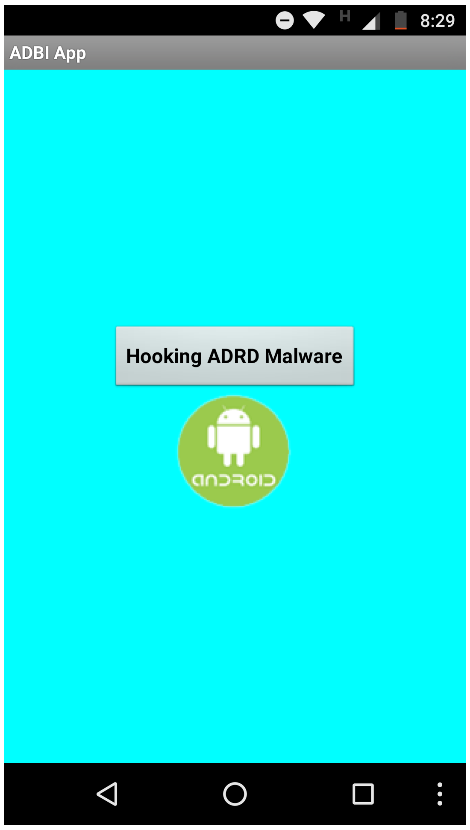


Figure 3.8: A malware sample instrumented by using the ADBI toolkit [95].

records of malicious URLs that could make mobile Apps in the form of malware. Here the key piece of all our infrastructure and where lies the logic of the presented method, it is the tool that implements the instrumentation of the Android application under examination. The data sources to be collected are, therefore, the traces with the invoked DNS queries performed by the malicious application executed, previously instrumented, in a controlled environment, by introducing hooks as described in [87]. The instrumented methods come from the Class "InetAddress" of the Android 4.3 API, and to complete the capture of the involved URLs, the hooking library called Android Dynamic Binary Instrumentation ADBI is also used [95]. This is because we have identified the Java Abstract class *URLConnection* (which cannot be instrumented at the level of the Android framework layer) as the main Java class used for opening the URL connections, and we recognize that in turn, this appeals to a native function of the lowest level called *connect()*. In short, we run the instrumented App under examination for a certain period of time, and it creates a text file with a list of captured URLs as explained later.

3.4.1.2 Second Phase: Identification of captured URLs

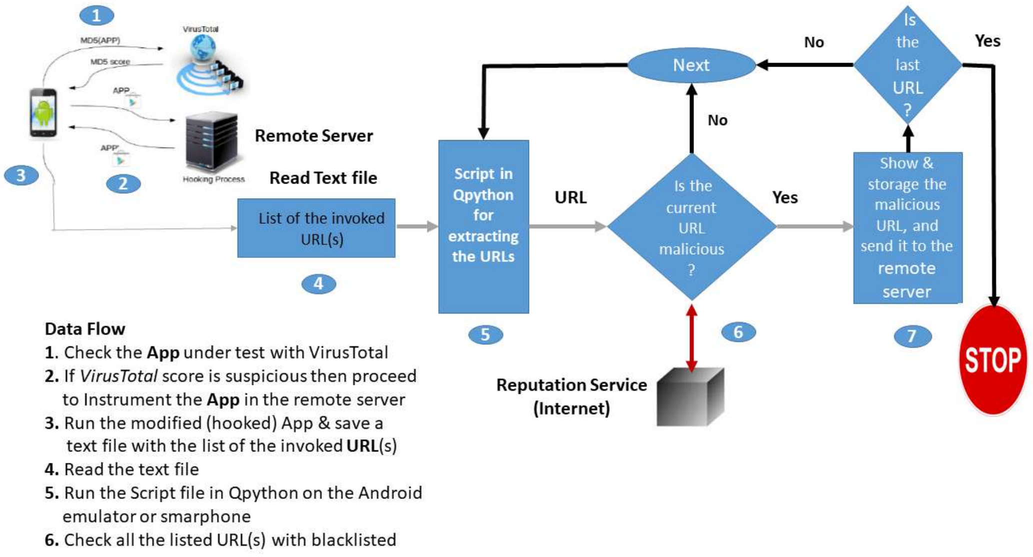
The objective of this phase will be to review and evaluate relevant sources of information security in the mobile network and show how they could be used to detect security events of users in the network. First of all, we can consult the trace database of the application using a QPython script written by us to extract the URLs involved ("qpython-language script"). So here we focus on the detection of malicious URLs. And for this we depend on several third-party APIs. With the help of these APIs such as URLVoid [98] and Web of Trust [93] we can know if the URLs obtained are in blacklists. These are free services that analyze a website through multiple blacklist engines or monitors and online reputation tools to facilitate the detection of fraudulent and malicious websites. This service helps them identify websites involved in malware incidents, fraudulent activities and phishing websites, etc. We consult known reputation grading services URLVoid [98] and/or Web of Trust [93] as aforementioned before. So, among URLs collected from experiments, we treated them as malicious URLs if both services said “malware”, as benign URL if both said “benign”. URLs with different answers are discarded.

3.4.1.3 Third Phase: Storage of malicious URLs

The objective of this phase will be to store in a text file (e.g., analysis.txt) and in a MySQL database the malicious URLs found, respectively, so that they are available for further analysis or at least notify the user of the smartphone of a connection to remote servers that can be considered dangerous.

3.4.2 Experimental Results

To evaluate our platform, we show the results obtained for several different stages of the complete process of monitoring the DNS queries invoked by the malware. First, we assume that we have already instrumented and monitored, and executed, one of the malware samples that performs DNS queries as it can be seen in Fig. 3.8, so we consider one of the ADRD Trojan specimen provided by MalGenome, and let it run for a period of 15 to 20 minutes on the Android Emulator of the Eclipse IDE; in order to capture URL queries requested. Besides, in Fig. 3.9, we have the whole block diagram used to detect malicious URLs.


1
MD5(APP)
VirusTotal
MD5 score
APP
Remote Server
APP'
Hooking Process
Read Text file
3
2
List of the invoked URL(s)
4
Script in Qpython for extracting the URLs
5
URL
Is the current URL malicious ?
6
Yes
No
Next
No
Is the last URL ?
Yes
Show & storage the malicious URL, and send it to the remote server
7
STOP
Reputation Service (Internet)

Data Flow

1. Check the **App** under test with VirusTotal
2. If *VirusTotal* score is suspicious then proceed to Instrument the **App** in the remote server
3. Run the modified (hooked) App & save a text file with the list of the invoked **URL**(s)
4. Read the text file
5. Run the Script file in Qpython on the Android emulator or smarphone
6. Check all the listed URL(s) with blacklisted

Figure 3.9: The processing of the URLs is illustrated for the proposed Non-root method for Dynamic DNS Monitoring of Android malware via instrumentation.

Thus, on one hand, we write a script in Qpython for extracting the URLs obtained by instrumented App. On the other hand, we now describe the processing of a single sample of malware used. The following steps illustrate by means of screenshots the proposed technique, regarding to the network traffic of the Android smartphone. Then, in the next screenshots of the Android emulator we show: (i) the computer language in use, Android python (Qpython); in Fig. 3.10 Qpython is running on the emulation environment, (ii) the Fig. 3.11 shows a way to execute the python script, in Qpython, and (iii) the Fig. 3.12, which presents the outcome of the evaluation of one of the samples of the ADRD Trojan malware and in the output of the script we have the malicious URL **addr.taxuan.net**, which is the address of the remote malware server.

3.4.3 Remarks on the Method2

In this section we propose a tool for Android malware detection termed *AppURL* based in instrumentation. However, so far, it presents some shortcomings because it has been not possible to implement an automatic toolkit version of the *AppURL*. This tool could contribute to develop a collaborative framework of that allows to find of events between the malicious URLs of the applications in the intelligent device, and the DNS records of the network traffic provided by the DSN Servers

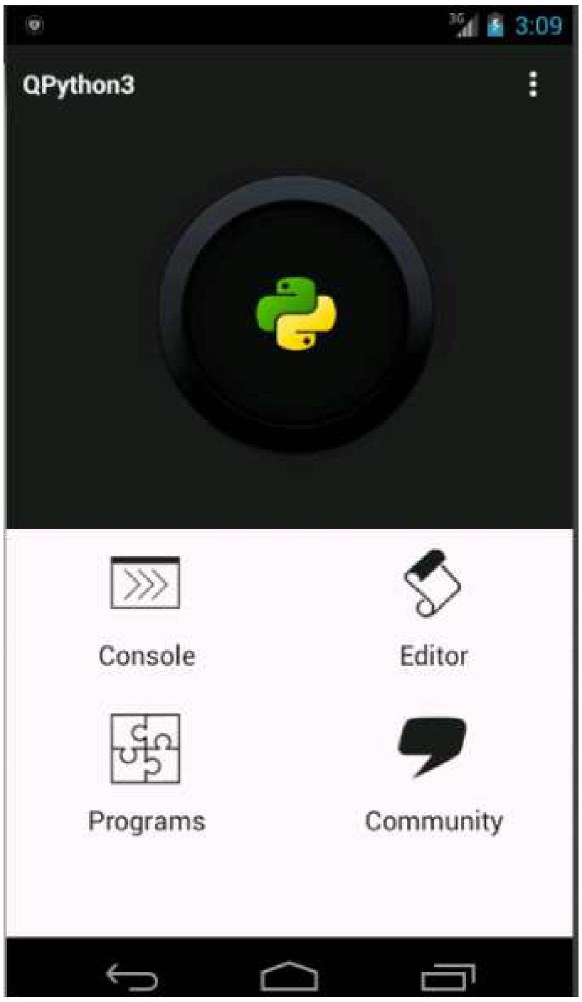


Figure 3.10: The App of the Qpython programming language up and running on the Emulator (Eclipse IDE).

at the mobile infrastructure. Here, we only work in the part that operates on the side of the smartphone, identifying URLs invoked through the DNS queries required by Apps under test (we want to know if they are malicious or not), in this case, samples have taken from the MalGenome dataset. This can be used to identify malware attacks on several smartphones that use the OS Android API level 19 without modifying the firmware or "rooting" it. These characteristics make it attractive to apply this method proposed here because of its practical implications. Furthermore, to be able to work with Android OS above API level 19, it will be needed to incorporate the Android ART VM into future research work, see [88, 89, 90, 91].

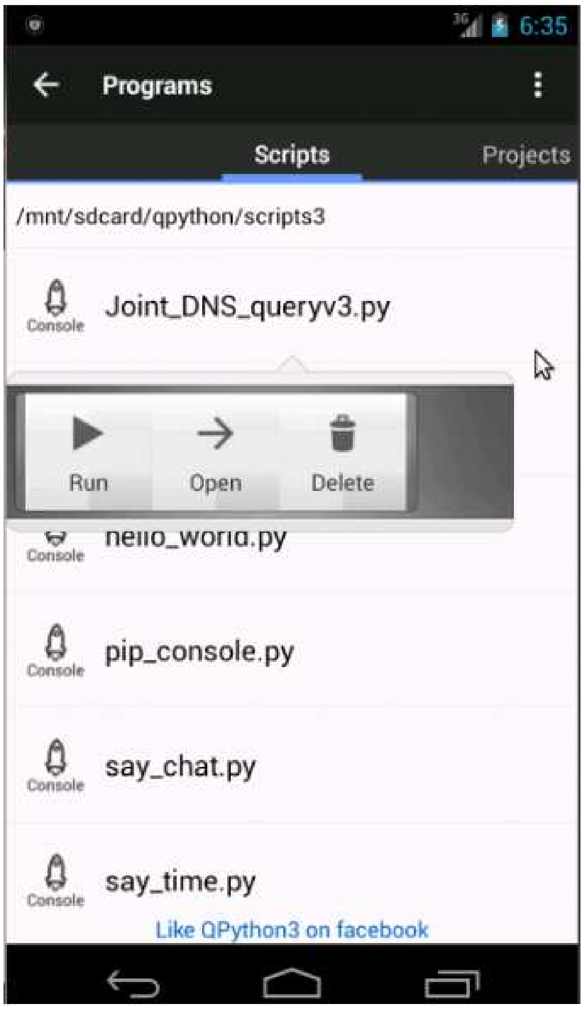
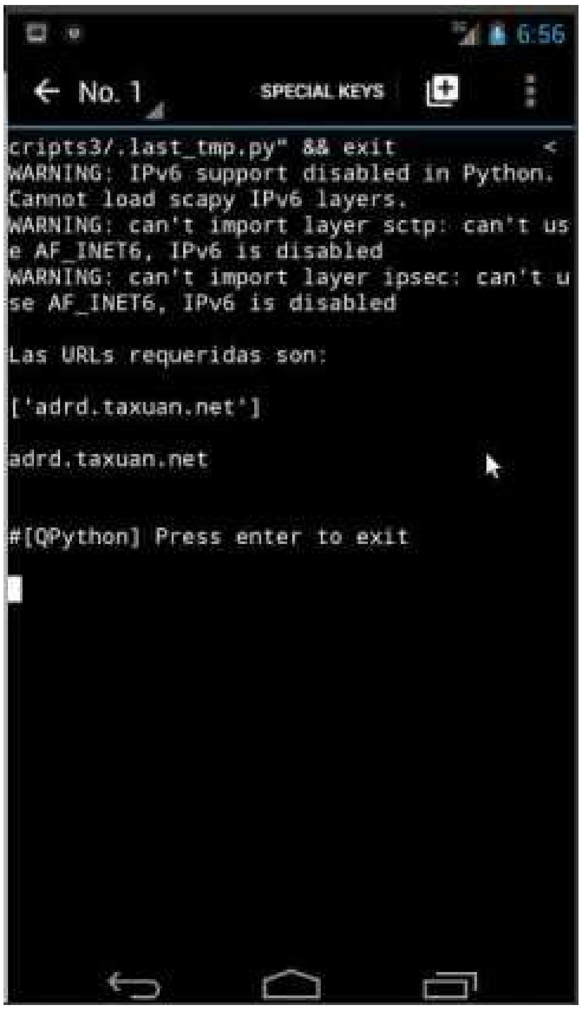


Figure 3.11: Executing the Script Qpython dubbed **JointDNSqueryv3.py**, on the Emulator (Eclipse IDE).

3.5 Comparison of the previous methods

Based on the aforementioned previous two sections, we proceed to compare **method1** versus **method2** as it is shown in Table 3.2 with regard to their (**Android version**), (**Automatic toolkit**), (**Battery Usage**), (**Memory resources**), and some comments on their pros and cons. Based on Table 3.2, we summarize:

- We need to have a real-time dynamic DNS request monitoring detection at the mobile device. So far, this is available with the **method1**.
- The most advanced feature regarding the Android versions is given by **method1**. As it can be seen in the Table 3.2, with **method1**, we examine App from Android version 5 and above. However, **method2** could be used for Android



```
cripts3/.last_tmp.py" && exit           <
WARNING: IPv6 support disabled in Python.
Cannot load scapy IPv6 layers.
WARNING: can't import layer sctp: can't us
e AF_INET6, IPv6 is disabled
WARNING: can't import layer ipsec: can't u
se AF_INET6, IPv6 is disabled

Las URLs requeridas son:

['adrd.taxuan.net']

adrd.taxuan.net


#[QPython] Press enter to exit
```

Figure 3.12: Results of the Qpython Script running in Figure 3.11 analyzing the trojan ADRD for tool termed *AppURL*.

version 4.3 and below.

- Also, with **method1** was possible to conduct tests for battery usage of the Apps. For **method2**, these tests were not done, since we conduct the experiments on the emulators. Also, note that the used Qpython scripts run very well on smartphones.

With the available results, we opted for the **method1**. This tool can be incorporated into a framework to detect malware by the DNS dynamic request monitoring of Android applications, integrating the smartphones and the DNS data of the DNS servers at the infrastructure of the mobile operators.

Table 3.2: COMPARISON of the Dynamic DNS Request Monitoring of Android Malware

Features	**Method1**	**Method2**
Android version	Up to version 4.3	From version 5.0 and above
Automatic tool	Yes	No
Batery Usage (on average)	0.05 %	Not available
Memory resources	Ok	Too many memory mapping
Environment Test	Emulators & Smartphones	Emulators & Smartphones

3.6 Limitations

Of course, one limitation of running experiments on emulators, it is the possibility that smart malware can detect this kind of environments and refrain from continuing the malicious attacks. However, it is tough to use only real smartphones for massive experiments.

3.7 Conclusions of the Chapter

Here we propose and discuss two methods to conduct Dynamic DNS request monitoring of Android malware. The so-called **method1**, which is based on the use of VPN-network traffic analysis at the smartphone to capture the DNS queries done by it. Also, we implemented the so-called **method2**, which is based on the instrumentation of the functions calls done by the target App at the smart device; the hooking process was done by using the ADBI toolkit [95]. For both methods, we were using well-known reputation grading services (blacklisted URL reputation services). Of course, blacklisted URL/IP services have several constraints regarding advanced malware attacks, so in the next chapter, we are going to propose a framework for Android malware detection more sophisticated that it takes into account the records of the DNS servers.

From the experimental results, we conclude that the most trustworthy method, it is **method1**, which is termed *Network Sentinel*. This method has publicly available in the Google Play Store for several months showing, so far, that is useful and it has an easy manner to be utilized.

Chapter 4

A Framework to detect Android Malware from DNS Servers

As was discussed in Chapter 3, Domain Name System (DNS) is the cornerstone of the almost all protocols and services of the Internet; and the way that most smart malware callbacks to their controllers' C&C infrastructure (often referred as to command and control server) in order to obfuscate their operation overall architecture from security monitoring. For a complete view of malicious activities, it is often necessary to enrich the DNS data by aggregating data from applications and networks from multiple sources, such as domain registration records and geographic location records of IP hosting domains, among others. So, having rich DNS traffic information is very important to identify malicious behavior and this research shall consider information sources that capture DNS traffic. In agreement with authors in [99]: "Several studies proposed using DNS for malware detection, because, it is the first step visiting a specific website". Here we propose a framework for DNS monitoring approach termed **S**ecurity **In** **M**obile **PL**atforms with **E**vent Analysis of **DNS** data (*SIMPLEDNS*).

4.1 Introduction

It is a well-known fact that the Internet is being used continuously to execute cyberattacks athwart different objectives. For instance, DNS plays a crucial role in network connectivity. Unfortunately, its open nature has made it one of the fastest growing vectors for malware threats.

As stated in [100]:"Benign services and protocols are being misused for various malicious activities: to disseminate malware, to facilitate command and control (C&C) communications, to send spam messages, to host scam and phishing webpages. Clearly, it is very important to detect the origins of such malevolent activities, be it by identifying an URL, a domain name or an IP address.”

Many approaches have been proposed for such purpose: network traffic analysis [101, 80], scrutiny of the content of web pages [102, 103], URL inspection [104], or using a combination of thereof. On top of these, one of the most promising directions relies on the analysis of the Domain Name System data.

Regrading the Domain Name Space, its primary task is to resolve requests for naming. This function could be explained as an analogy with a telephone information service that has current contact data and facilitates it when someone requests it. Thus, the domain name system uses a global network of DNS servers, which subdivide the namespace into managed areas independently from each other. This system allows a distributed management of domain information. Each time a user registers a domain, a **WHOIS** entry is created in the corresponding registry and stored in the DNS as a "resource record." The database of a DNS server becomes, thus, the compilation of all records in the area of the domain namespace that it manages.

Furthermore, DNS protocol is a fundamental part of the Internet, which is the way that easy, memorable domain names are localized and translated into the Internet Protocol (IP) addresses. The domain name system maps the name people use to locate a website to the IP address that a computer uses to localize a website. So, the detection of malicious domains through the analysis of DNS (data) logs have several benefits compared to other approaches. First of all, DNS logs constitute only a small fraction of the overall network traffic, which makes it suitable for analysis even in large scale networks which cover large areas. Moreover, caching, being an integral part of the protocol, naturally facilitates further decrease the amount of data to be analyzed, allowing researchers to explain even the DNS traffic coming to Top Level Domains [105]. Second, DNS traffic contains a significant amount of essential features to identify domain names associated with malicious activities. Third, many of these features can further be enriched with associated information, such as Autonomous System number (ASN), domain owner, etc., providing an even more precious space exploitable for detection. A large number of features and the vast quantity of traffic data available have made DNS traffic a prime candidate for experimentation with various machine learning techniques applied to the context of security. Forth, although the solutions to encrypt DNS data like DNSCrypt [106] exist, still a significant fraction of DNS traffic remains unencrypted, making it available for the inspection in various Internet vantage points. Last but not least, sometimes researchers can reveal attacks at their early stages or even before they happen due to some traces left in the DNS data.

Besides, DNS Protocol was initially being designed with no security protection in place. Subsequent The Domain Name System Security Extensions (DNSSEC) added a layer of trust on top of DNS by providing authentication and message integrity whilst remaining backwards compatible, but it still did not address issues such as Denial of Service (DoS)/Distributed DoS (DDoS) attacks and deployment difficulties. Yet despite the fact that DSN is vulnerable to a variety of attacks, which have been well known since the late 90s, there has been very little adoption of DNSSEC. Blockchain technology offers an innovative perspective to tackle those challenges and it has been proposed for the next generation of DNS [107].

The purpose of this chapter is to describe an approach that aims at detecting domains involved in malicious activities through the analysis of DNS data logs. The first observation we have made is that this research area is relatively new. Here, we focus on passive DNS techniques. The seminal paper [108], which led to the area as we know it today, dates back to 2005. This work was the very first published paper not only to consider using DNS records to detect malicious domains but also to propose a practical solution to obtain large amounts of data amenable to various types of analysis. A detailed survey on Malicious Domains Detection through DNS Data analysis is presented in [100], where the authors propose a general framework to describe the various components required to implement a DNS based detection technique, namely: Data Sources(DNS data collection (Where are the Data Collected: a)Host-resolver b) DNS-DNS) and How are the Data Collected (active and/or passive), Data Enrichment(Geo-location, ASN, Registration records, IP/domain black-/whitelists, Associative resources records, Network information), ground Truth), Approaches or Design of detection algorithms(Features, Detection methods (Knowledge based, Machine learning based, Hybrid), outcome(agnostic, specific)), Evaluation methodology (Metrics(e.g., type of metrics utilized in machine learning), evaluation strategies) .

4.1.1 Problem Statement

This work aims to deal with the sophisticated and emerging threat of Android malware in mobile ecosystems. We develop techniques to systematically explore and monitor the App traces generated from the execution of Apps via the capture of the network traffic, after that they are sent to the cloud service to support the malware detection. Much of the research work surrounding mobile malware has been centered on either the in-depth analysis of malicious Apps (host level) or the network-based approach (network level). Of course, developing a collaborative framework between the former methods seem to a natural step, and it can increase the chance of malware detection. Concretely, here, we focus on the discovery of malicious URL through DNS Data Analysis. The main goal of this research is to explore, design, and develop techniques that can be used to detect malicious mobile behavior from large sets of heterogeneous sources. In particular, the DNS-service network traffic activity produced by mobile malware will be inspected and correlated with device-related activity.

4.1.2 Contribution and Outline

The contributions of this chapter are (1) extending the system for Android platforms described in [60] composed of implementation on the smartphone side and the remote server side. Herein, we focused on capturing the requested URL for detecting malicious transactions initiated by an App running on the Android phone; and (2) evaluation of combining and correlate the following two approaches: top-down detection by identifying malware domains using DNS-service network traffic and bottom-up exposure using the classical Dynamic Analysis (DA) on a number of Apps to pinpoint the malware.

It is worth to mentioned that App *Network Sentinel* has been updated to the version 1.1 in Google Play Store since March 2019 to support the interconnection with relational databases.

This chapter is organized as follows. In Section 4.2 we provide the related work and necessary background of the framework to detect Android malware from DNS servers; as well research methodology, while in Section 4.3 we have the Results of the computational experiments carried out in this chapter.In Section 4.4 we discuss the limitations of the proposed DNS-based framework and in Section 4.5 we have the concluding remark of this chapter.

4.2 The Framework to detect Android malware using DNS Servers

As aforementioned in the previous chapters, in recent years, Android malware has been considered one of the significant security issues and fast-growing threats facing the Internet in the mobile arena. At the same time, DNS is widely misused by miscreants to provide Internet connection within malicious networks. Here, we apply an infrastructure for monitoring the Android applications in a platform-independent manner, which is based on dynamic DNS request monitoring of Apps via networking. These traces are collected at a central server where string pattern matching is used, machine learning algorithms are applied, and visualization takes place. From these traces (Apps logs) we can extract URLs and correlate them with DNS records, enabling us to find the presence of malware running at the network level; either through the usage of blacklisting or machine learning techniques. Example of the fields of the database that we use to store the App traces are: *timeStamp*: Time in which the DNS query request was made, *packageName*: Name of the package which helps to identify the application, *aplicationName*: Name of the application, *phoneID*: Phone identifier, *cURL*: captured URL, and *ipSource*: IP used by the smartphone.

Many security mechanisms were proposed to detect mobile malware and protect targets from attacks. In general, most of these mechanisms are based on analyzing App elements such as permissions, the used application programming interface (API) function calls, the employed system calls, or its bytecode. Such mechanisms employ various detection techniques such as static dissection, dynamic analysis (DA), and cloud-based analysis. In the static analysis, there is an attempt to identify the malicious code by decompiling/disassembling the App and searching for suspicious strings or block of code. The DA implies the execution of the App performed through instrumenting or virtual machine monitoring to observe its behavior. In the cloud-based approach, the App will be executed and dissected on a remote server.

Mobile devices have become significant targets for smart malware due to their substantial network activity, including Internet access. So, DNS is one of the critical elements of the Internet that facilitates associating a domain name and hosting IP address. Besides, the DNS scheme is a query/reply based protocol where the authenticity of the response is not confirmed or confirmed by approaches that can be thwarted easily. However, in addition to the crucial role in the functioning of the Internet, DNS is extensively misused by malware developers. Thus, the aggressors rely on DNS to provide adjustable and resilient communication between compromised client machines and malicious infrastructure. However, it is worth noting that we do not address or detect malicious DNS in this work, which is DNS traffic corrupted for illicit and malevolent reasons. In fact, we only take advantage of DNS to find malware without having to monitor all smartphones in a system.

This work focuses on monitoring Android applications' suspicious behavior at runtime, in particular integrating to the App traces (described in [61, 58]) of the captured URLs requested to remote servers by the App of the smart device. Later, we correlate these enhanced App traces with DNS traces taken from the DNS servers of the mobile infrastructure. Thus, we propose a platform-independent, dubbed *SIMPLEDNS*, behavior monitoring infrastructure. It is composed of four elements with the capacity of capturing the DNS queries requests by the App under test via traffic network monitoring. To do this, we utilize an ad hoc packet sniffer [58] publicly available at the Google Play store and developed by us. *SIMPLEDNS* is composed by: (i) an App (sniffer) that guides the user in selecting and monitoring of the application to be examined and it sends the detected URLs to a processing server, (ii) a cloud (processing) service that collects the App traces and it has the capability to classify the URLs by a module of machine learning, (iii) the DNS servers that provide data logs of the DNS network-service traffic, and (iv) finally a cluster of Elasticsearch technology [109]. The *Elastic stack* includes a visualization component that can generate dashboards of the top-ranking classification of URLs based on *Kibana* (part of the Elastic Stack, it is an analytics and visualization platform that builds on *Elasticsearch*(*ES*) to give us a better understanding of the data). Also, the DNS records are sent the input data (DNS logs) to the tool dubbed *Logstash* (an agent and server-side data pipeline processing that receives it, parses it and later sends the indexes into *ES*). The log files of the DNS data are analyzed by using *Elasticsearch* technology, namely the so-called *Elastic* stack (*Elasticsearch*, *Logstash*, and *Kibana*). An Overview of the monitoring system is shown in Figures 4.1 and 4.2. See further details about our infrastructure in section 4.2.2.4

4.2.1 Related Work

Since most of the Android malware resort to communicate with some remote server (e.g., a botnet master machine), there is the crucial need to detect fraudulent or malevolent operation with the help of a collaborative malware analysis framework between the smart device and the network traffic involved. Besides, usually, the malware analysis comprises the process of reviewing the code and gets information about the behavior and functionality of the malicious software in its environment. Afterward, the results of the analysis will be used as an input to the Malware Detection (MD). The type of analysis for identifying malicious applications in the Android platform can be classified as follows, namely: Host-based

Analysis and Network-based Analysis. Here, we will be focused on the latter. The so-called smart malware in current smartphones and tablets have mushroomed over the last few years, which is supported by sophisticated techniques intentionally designed to master security architectures in use by such devices. Let us review some of these following approaches, namely: Network-based Android MD and Collaborative Approaches for Android MD.

4.2.1.1 Network-based Android MD

Several approaches explicitly analyze network traffic for different goals. In [80], they address the network-based malware detection mechanisms for Android-based attacks, and they use *MalGenome* [36] dataset in their research. So, the authors used four different traffic categories (network traces), namely based on: DNS-based features, HTTP-based features, Origin-destination based features and TCP-based features. This trait analysis is used to train a detection app model for classification of Apps based on ML algorithms. Furthermore, in *CREDROID* [101], it has been proposed an Android malware detection by network traffic analysis capturing packets in a remote server using the protocol analyzer WireShark [85]. They also introduced the reputation score of the URL. With all of this, the authors proposed a method which identifies malicious Apps on the basis of their DNS queries and APK score computation through Virustotal [110], as well as the data it transmits to remote server by performing the in-depth analysis of network traffic logs in offline mode. Unlike [80] and [101], we profile only gathering the network traffic of the app under test at the smartphone side, without rooting the Android phone. Most of the published host-approaches do not integrate the network traffic dimension at the device side into the analysis. But, unlike the component-application analysis, we include the DNS traffic in our approach at the network level of the smartphone. So, to attain the goal of detecting the malware, we propose a dynamic inspection combining the DNS queries at the Android phone level (app traces) and the DNS log files from the network operator at the infrastructure level.

4.2.1.2 Collaborative Approaches for Android MD

Han et al. [111] proposed to identify malicious Apps by analyzing malware traffic on the mobile Internet and achieved a high detection rate and scalability in their system. Specifically, the authors designed a real-time Android malware detection system based on network traffic analysis and distributed third-party scanning services. This system is composed of a training model and a real-time detection model. By training over the malware traffic (they capture malware samples traffic, then used the distributed third-party scanning services to get malicious URLs) using the training model, they found that 76.33 % DNS queries and 45.39 % HTTP requests are all malicious. By performing malware detection using the established real-time detection model, they showed that the detection rate using the real-time scanning service is much higher than the integrated service. Meanwhile, the detection rate will further improve by integrating more third-party scanning services into their system.

In [16], the authors suggested a method for identifying compromised clients based on DNS traffic analysis combined with graphs. As it is well-known Internet criminals misuse to support communication within their malevolent network infrastructure. Besides, to evade traditional detection approaches based on domain and IP blacklists, attackers resort to the so-called agile DNS mapping or dynamic DNS techniques (e.g., Fast-flux and Domain-flux). These techniques involve swift changing of domain names or/and IP addresses associated with a single fully qualified domain name for malicious servers. This work targets both Fast-flux and Domain flux, thus having an advantage over current detection methods that identify infected clients based on DNS traffic analysis in large-scale operational infrastructure, from different Internet Service Providers networks.

Additionally, in [13] a collaborative framework for characterizing malicious behaviors on Apps is presented here by using the following features: (i) network patterns or usages (ii) host domain reputation with the App is connecting to, (iii) which APIs are used and (iv) which permissions are used. Thus, they have designed a detection system based on these features by implementing four engines, namely: network behavior analysis engine, host domain reputation analysis engine, critical API call pattern analysis engine, and Android permissions use analysis engine. Each engine then monitors its particular trait from Apps and independently detects malicious behavior based on ML techniques. So, each engine makes its decision, and the given information from four engines are correlated into a final decision. In other words, the correlator determines the ultimate decision. The tests conducted with thousands of Apps had proved that detection with this approach can be reached with a very low rate of error (e.g., a precision rate of the final decision of 91.25% is achieved using Support Vector Machine (SVM) and with very little overhead.

Our proposed infrastructure is related to some of the research work mentioned above and employs similar traits for identifying malicious applications, such as

DNS queries, algorithm design, and DA. However, our approach is different from the aforementioned approaches in the following aspects. Firstly, we have a runtime malware detection (dynamic analysis) but abstain from reshaping the firmware or rooting the smart device as it is done by [40, 112, 13]. Also, the DNS queries requested from the Apps under test are captured at the smart device, but not on a remote server using **Wireshark** as it is done in [80] and in [101]. Secondly, we combine in a collaborative or integrated environment the bottom-up analysis (Network-Level monitor at the smartphone) with top-down approach (DNS-service network traces) in an easy-to-follow manner in the cloud service and *ES* cluster. Thirdly, moreover, we are able to monitor almost in real time, not just the DNS queries request for a particular app to be monitored, but we are able to focus on intercepting malicious URLs at the traffic network affecting others smart devices. Our platform is more dynamic and collaborative than the other approaches mentioned above.

On the other hand, for instance, let us focus Android botnets regarding the capture of theirs Command and Control URLs. We are wondering: How many URLs are requested by a malicious app using dynamic analysis (DA)? In this case, DA usually compels a botnet sample to reveals hidden URLs. Let us review one of the previous works by authors in [113], where they proposed a method to detect Android botnets; first, they collected a dataset of 19129 malware samples (including some of them from MalGenome project) comprising 14 Android botnet families, their characteristics and communication behavior. Second, they extracted all the hidden URLs within these families through static and dynamic analysis. These analyses helped them to illustrate and visualize the C&C communication patterns of android botnet applications. Their experiments with some malware samples show that there are different types of relationships (one-to-one (one APK file is associated with a unique URL), one-many (one APK file are associated with several URLs), many-to-many (many APK files contain many URLs)). In this study, they showed that some samples of the malware *AnserverBot* are utilizing public blogs to set up their C&C URLs to send commands to bot clients. So they managed to obtain from public blogs up to 830 C&C URLs (8 unique URLs) from 244 APKs, that adopted many-to-may relationships. Some of malevolent domain names detected are: *91.cookier*, *baisu.com*, *b4.ccookeier*, *sina.com*, *b3.8866.org*, among others. Another interesting finding of the Command and Control URL pattern is on the DNS. They found that the C&C URLs exploit its DNS by adopting the Domain

Generation Algorithm (DGA) and the URL obfuscation techniques [114] (e.g, obfuscating the host with an IP address, obfuscating with the large hostname, and unknown or misspelled domain). The latter result is very interesting because we can utilize heuristic rules based on lexical analysis [99] of the URLs to further extend the malware detection capabilities provided by the blacklistings.

Our results could be shown in a dashboard that visually render existing malicious URLs in the system enabling to warning a potential mobile operator about their presence in its traffic network. Noting that a mobile operator can easily or indirectly detect other infected devices that had not installed the monitoring application. This is due to they behave in the same manner, by doing the same DNS queries than the monitored devices. This is certainly a very valuable benefit because we do not need to monitor all the smart devices at the same time. Since we collect the used URLs on the Android device instead of on a remote server or gateway, we shorten the time to detect malware as it is suggested in the hybrid analysis method dubbed *NeseDroid* [17].

4.2.2 Research methodology

Obviously, once an App visits a malicious URL, it may become a malware. So, malware refers mainly to the software with malicious behaviors running in a host or a network system. In this work, we come up with or conceive a method that uses the URLs visited by Apps to identify malware. Hence the whole process is divided into four phases. First of all, the first phase implies the generation of the app traces, data collection and the analysis and monitoring of network traffic. Actually, the app traces are, in this particular case, the malware traces with the plus in this approach that we are able to capture the DNS queries done by the app under test if any. The second phase includes the log aggregation and transport of data generated, the extraction of URLs from app traces and DNS-service network traces are done with the help of Python-language scripts. The third phase is the search and analytics task (without and with Machine Learning algorithms). And, the fourth phase is the visualization component of the system. The rest of the process is outlined in subsections that follow.

An overview of the monitoring system are shown in Figure 4.1 without machine learning (ML) and in Figure 4.2 with ML.

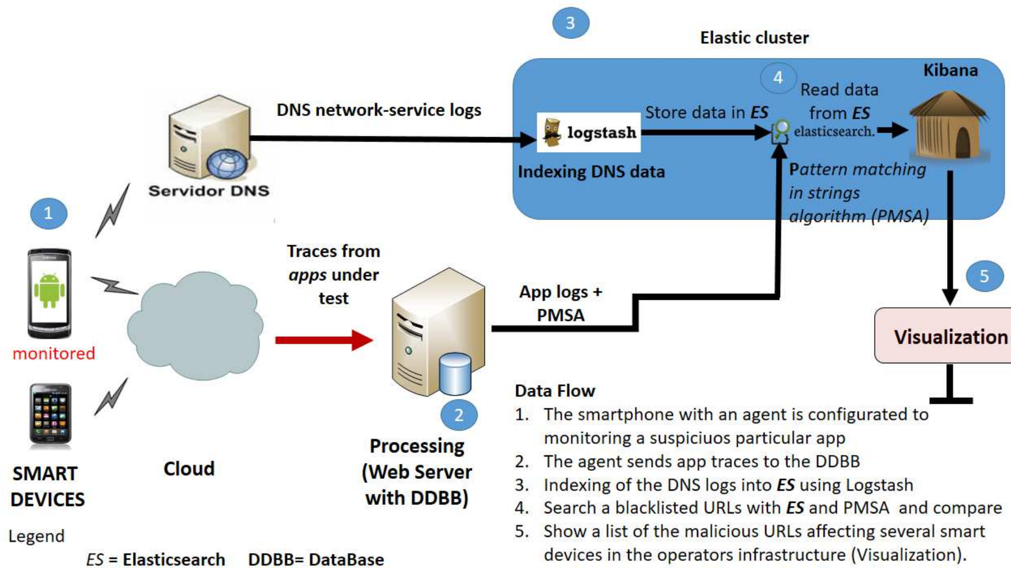

3
Elastic cluster
4
Read data from ES
Kibana
DNS network-service logs
logstash
Store data in ES
elasticsearch.
Indexing DNS data
Servidor DNS
Pattern matching in strings algorithm (PMSA)
1
Traces from apps under test
App logs + PMSA
5
monitored
Visualization
2
SMART DEVICES
Cloud
Processing (Web Server with DDBB)
Data Flow
1. The smartphone with an agent is configurated to monitoring a suspiciuos particular app
2. The agent sends app traces to the DDBB
3. Indexing of the DNS logs into ES using Logstash
4. Search a blacklisted URLs with ES and PMSA and compare
5. Show a list of the malicious URLs affecting several smart devices in the operators infrastructure (Visualization).
Legend
ES = Elasticsearch
DDBB= DataBase

Figure 4.1: Proposed approach without ML. "Servidor DNS" means DNS Server.

4.2.2.1 Introduction to the Malware Dataset

First, let us introduce one of the employed Malware Datasets. In our experiments, we partially are using the MalGenome dataset [36], which it has 1260 Android application package (apks). So according to our approach, we need to "capture" the network traffic from 100 malware samples of these applications (apk files). It should be mentioned that there are malicious Apps that are not generating network traffic, therefore these cases are not taken into account or ignore in our analysis.

4.2.2.2 Data Generation (DNS Data Sources)

This part is done in two flavors, namely: i) DNS generation from MalGenome dataset, and ii) DNS Data from the DNS servers & and app dubbed *Network Sentinel* [58].

Part1: DNS Data from a list of malicious (MalGenome) and benign Apps used for data collection. Here we are interested in the data sources that will feed into our platform, namely: the app traces from smart devices conveying information about the DNS consultations done by one app under test; and the DNS-service network traffic in the mobile infrastructure, in particular, the logs from the DNS servers. To achieve our objective, we utilize the Android OS Version 5.0. In order to collect data at the smart device level, we need to set up an experimental testbed

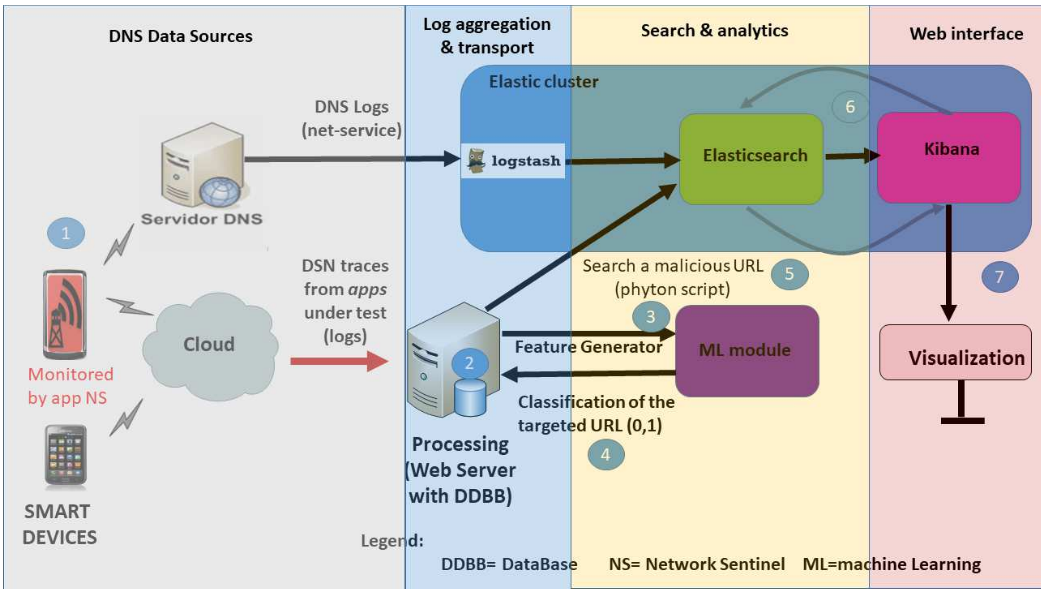


Figure 4.2: Proposed approach with ML.Data Flow: 1) Monitored URLs from Apps collected by *Network Sentinel 1.1* and send to processing server, 2) Traces from the Apps are saved into the DDBB, 3) The processing server computes the vector features to feed the ML module, 4) The ML algorithm classifies the URL under scrutiny, 5) If the URL in 4) is found malicious then a Python script is utilized to conduct a search based on ES and figure out how many smart devices are affected by the malware, 6) Kibana shows some malicious URLs found in the DNS-service network records under scrutiny. "Servidor DNS" means DNS Server.

with multiple virtual machines (VM) which used VMWare WorkStation 12 and VirtualBox 5.0.28, respectively; to create a controlled environment. Afterward, we use either the smartphone or the *Android Studio Emulator* on a host a machine which is employed for running the Apps and the App Network Sentinel; and we then run various tools to process the malicious Apps under analysis, in particular in the Virtualbox VM we run the *Elastic Stack* [109]. Let us introduce the first data source, the processing of the traffic generated by the smartphones in the VMWare VM. Thereafter, we store some privacy data (e.g., contacts information, images, and some downloaded files) to the emulator, the next step performed is to capture the DNS queries of the samples from each MalGenome family in use. Samples from each of the malware family were executed on the emulation environment for a short and fixed amount of time (10 mins). We expect some of the samples to communicate to the remote server since each sample itself is a malware. To separate network traffics of the smart devices, the virtual machine is left idle for around 5 minutes between running and terminating of applications after the network traces from the App under examination are captured and saved. After the traces have been collected from an application, it is uninstalled and *Android Studio Emulator* is rebooted. It is important to mention that the aforementioned procedure, it is just to check out which malware samples are connecting with remote servers. Also, we are to generate DNS traces from the smart devices by monitoring with a tool named *Network Sentinel*, for further details see [58]. The benign Apps consider, for sake of comparison, in this study are (all of them have been taken from Google Play Store): Book Read, Dictionary.com, BBC News, Maps, Facebook, WhatsApp, Email, Youtube. On the other hand, the malicious Apps [36] involved are the following: ADRD, Anserver, BaseBridge, DroidKungfu 1, DroidKungfu 2, DroidKungfu 3, DroidKungfu 4, Geimini, PjApps, Plankton, RougeLemon, DroidDream, DroidDreamLight.

Part2: DNS Data from the DNS servers & and Apps under test. Regarding the second data source, for sake of simplicity, in this work; we initially only consider two samples of DNS-service network logs (each file has around 30 MB in size, which are currently available for our experiments) provide by one mobile operator in 2015 and 2016. DNS records contain valuable information about domain names and associated IPs that clients query, whether mobile or not. It will therefore contain usually malicious URL records that could make mobile Apps malware. Later on, we will utilize some DNS logs collected from the main campus of the National University of Engineering (UNI) in Managua, Nicaragua. Typical sizes of these logs are around 3 MB (2019).

4.2.2.3 Log aggregation and transport

This task will focus mainly on the description of the agent responsible for collecting and storing information on DNS-service network logs. For data collection, the data will be used in text format from the DNS Servers, and DNS data (traces) collected in a well-structured relational database of the Apps using *Network Sentinel*. The collection of the logs of the DNS servers will be indexed through *Logstash* [109], see Figure 4.3. This tool, designed to collect and add events and logs created on multiple devices and services, sends information from DNS-service network logs to a system that indexes content for durable storage. Here, the main idea is to be able to have a telemetry correlator that will provide the analysis and correlation of all device telemetry data with all the sources that will be available to the system. Thus, on the other hand, when malware is executed, malicious URL

que the
clou

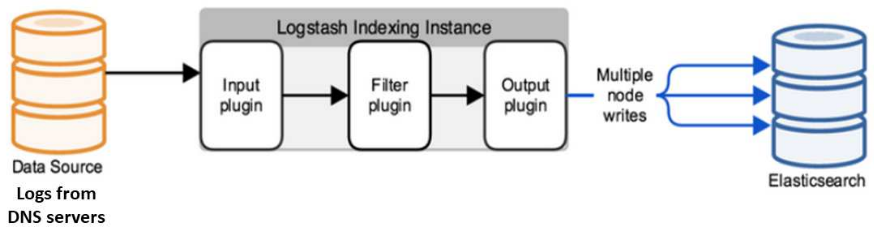


Figure 4.3: A Logstash instance has a fixed pipeline constructed at startup, based on the instance's configuration file. Picture credit: Deploying and Scaling Logstashhttps://www.bogotobogo.com/Hadoop/ELK/ELK_ElasticSearch_Logstash.php.

4.2.2.4 Search and Analytics

In this task the correlation of the monitored events in the previous task has to be carried out without or with ML.

Let us begin with the case without ML. Here the utilized strategy is to index the content of data collection in order to develop a pattern matching system, in particular, we do that with the DNS-service network logs that are expected to be a huge amount of data. Thereafter, we proceed with the extraction of the URLs included in the app traces of a particular sample of malware in use, if there is any URL. Malicious labeling through the usage of blacklisting could be also done by the *Network Sentinel*. Next, the following step is conducting a search of malicious URLs on the aforementioned content indexing built in with *ES*. So, we are able to identify other smart devices who are running the same malicious URLs previously executed. By the way, here *ES* has been used as a method of content indexing. *ES* is a Lucene-based search server [109]. Also, our system is designed with an easy-to-use web interface. The connection between the processing server and *ES* is supported by a Python script. The *ES* makes simple to search and perform various forms of analysis on the Apps and their traces, as well as the DNS-service network traces from the infrastructure of the mobile operator. In our case, we search for pattern matching in strings for those common malicious URLs found both in the app traces of the smart devices and in the DNS-service network records (logs) extracted in the network traffic of the mobile operator, see in Figure 4.4 without ML.

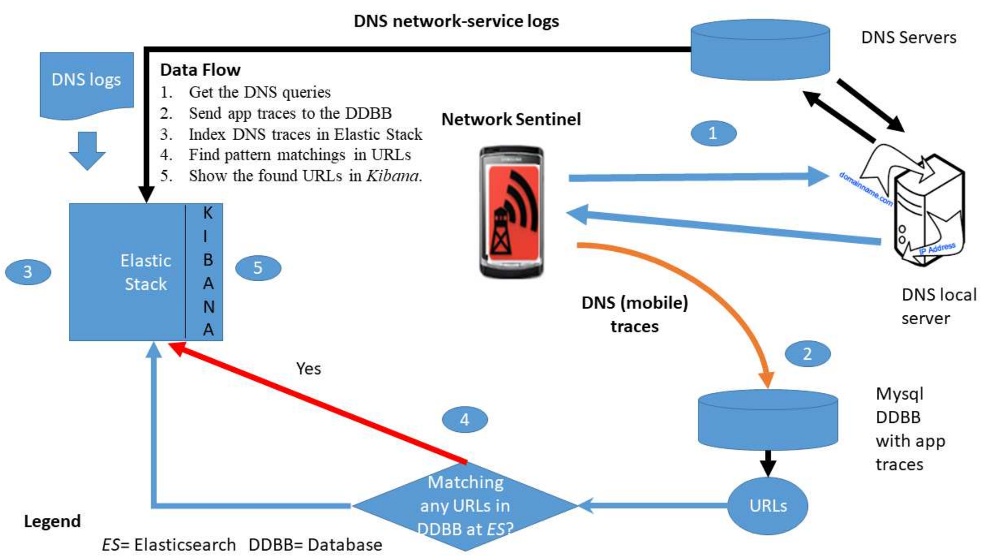

DNS network-service logs
DNS Servers
DNS logs
Data Flow
1. Get the DNS queries
2. Send app traces to the DDBB
3. Index DNS traces in Elastic Stack
4. Find pattern matchings in URLs
5. Show the found URLs in Kibana.
Network Sentinel
1
domainname.com
IP Address
Elastic Stack
KIBANA
3
5
DNS (mobile) traces
DNS local server
2
Yes
Mysql DDBB with app traces
4
Matching any URLs in DDBB at ES?
URLs
Legend
ES= Elasticsearch DDBB= Database

Figure 4.4: Correlation of the blacklisted URLs from the smart devices traces obtained with the App name *Network Sentinel* with the logs from the DNS servers without using ML.

The DNS server logs are usually huge in size, which is very difficult to analyze simply by looking at the log files. Then, we exploit the *Elastic* stack to find quickly the fields of the DNS records. The main advantage of *Elastic* is that it integrates search capabilities and visualization. Since the *elasticsearch* is highly scalable, in principle, it can search in "any" data size. Later on, we will come back to discuss this assumption.

In the sequel, let us now continue with the case of Search and Analytics task with ML. In general, blacklists provide robust evidence about blacklisted domains. Even though reputable blacklistings are the first line of defense, however, they still have a number of troubling issues. They cannot be exhaustive and none of them is wholly reliable. Thus, we need extra more accurate security measures and clearly, ML techniques are a promising line of action, so that must be put in place to ensure more intelligence is used to protect the users.

Furthermore, sometimes blacklists can exhibit high false positives and false negatives rates as well as they cannot cope with the rate at which newer malicious domains reveal themselves due to fast-flux services. Therefore, we can resort to integrate into the case of *Search and Analytics* phase, the ML algorithms. So, in order to refine the generality capability of malicious URL detectors, several ML techniques have been explored [115]. See Figure 4.2.

Here, we utilized Supervised Learning. Because in this technique we have knowledge about the dataset, mainly understanding about the correct output [58] of the algorithm and its relation to the input. The types of problems that are covered are regression and classification. We focus on the second type of problems, namely, classification. Next, the ML algorithms of the scikit-learn 0.20.1 library that were used during the study are: Support Vector Machine (SVM) and Random Forest (RF). Sahoo et al. [115] provide a comprehensive and thorough survey, that reviews the most common types of ML algorithms utilized to detect malicious URLs. In section4.3, we describe the proposed Framework with ML (*SIMPLEDNS*) in more details.

First of all, in order to build a detection model ad hoc [99], we collected two kinds of domains (malicious and legitimates) for reputation train. So, on the one hand for the training set construction, we need sample domains. The database of benign/malicious domains is collected from the URL dataset (ISCX-URL-2016) of the Canadian Institute of Cybersecurity (https://www.unb.ca/cic/datasets/url-2016.html). Over 35,300 benign URLs were collected from Alexa top websites at the following URL: https://www.alexa.com/. Also, malware URLs are more than 11,500 URLs related to malware websites were obtained from DNS-BH[1] (Malware Domain Blocklist by RiskAnalytics) which is a project that maintains a list of malware sites (in this work we chose 5000 malware domains and 4500 legitimate domains). Again, those are further used for training the classifier, then the SVM and RF classifiers generated a model, respectively. Finally, we labeled the Dataset: 0 for benign, and 1 for malicious.

Second, regarding the features selection, we are going to use a slight variant of the feature set proposed in [99, 45]. The features are as following:

- FD1: The length of the domain name
- FD2: The number of dots found in the domain name
- FD3: The number of hyphens found in the domain name
- FD4: The number of numerical characters found in the domain name
- FD5: Entropy of the URL

[1]https://www.malwaredomains.com/

- FD6: Suspicious top-level domains-based features
- FD7: Inappropriate words and transfer-based words found in the domain name
- FD8: Days elapsed since registration
- FD9: Average TTL value for the domain.

In short, the Domain Name-based Features are: Basic features (Number of characters (usually on average 12-13 characters are a good sign of a benign domain name), Number of dots (more than 3 dots in the domain name is related with malware with high probability), Number of hyphens(the number of hyphen in the domain name because benign domains have at most two hyphens), Number of numerical digits, list of suspicions Top Level Domain, and Tokens (Inappropriate words and Transfer-based words). We also add three more features such as **whois** query, the Shannon entropy of the string of characters present in the URL, and the average Time-To-Live (TTL). Afterward, we run a third python script to read an URL from the BBDD (processing server) and extract the chosen features aforementioned (feature vector) that are handled to ML algorithm. Again, the first python script allows us to search for URLs in the *Elastic stack*. The second python script enables us to compare the searching time of the URLs inside *ES* versus the searching time directly on the DNS Server log file through the KMP algorithm [116]). Those characteristics will be subsequently used in the training and testing process.

4.2.2.5 Visualization

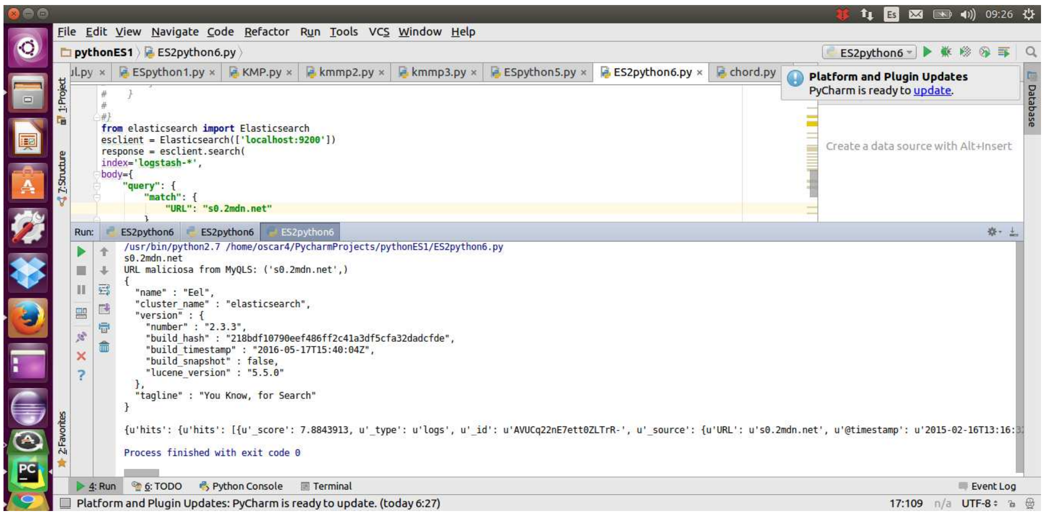
The visualization of anomalous behavior is the last component of the proposed architecture. In order to perform a visual analysis of the platform. So, we use the Elastic stack, which is a versatile collection of open source software tools that make gathering insights from data easier [109]. Formerly referred to as the ELK stack (in reference to *ES*, *Logstash*, and *Kiban*a). In particular, *Kibana* is a browser-based or web-interface visualization frontend for *ES*. It enables users to easily consume data in aggregate that would otherwise be difficult to process; making logs, metrics, and unstructured data searchable and more usable for humans. So because *Kibana* persists most of its data within *ES*, managing *Kibana* dashboards and visualizations is a similar exercise as managing other indexes in *ES*. Charts, graphs, and other visualizations sit atop *ES* APIs which can be easily inspected for closer analysis or use in other systems.

4.3 Experimental Results

The assessment of the proposed framework is two folded, namely: a) Without ML and b) With ML.

4.3.1 Experimental Results without ML

To evaluate our framework, in this subsection we show the visualization results of the process of monitoring the malware behavior. Firstly, let us have monitoring and running our DNS sniffer, one of the malware samples that it does DNS queries, and we let it runs for a long time. By the way, from the 100 sample families explored, only 63 of them have connections with at least one remote server. We then proceed to apply the pattern matching in strings by using Python-language scripts developed for this purpose (see Fig. 4.4).


File Edit View Navigate Code Refactor Run Tools VCS Window Help
pythonES1 ES2python6.py
ul.py ESpython1.py KMP.py kmmp2.py kmmp3.py ESpython5.py ES2python6.py chord.py
Platform and Plugin Updates
PyCharm is ready to update.
Create a data source with Alt+Insert
Run: ES2python6 ES2python6 ES2python6
4: Run 6: TODO Python Console Terminal Event Log
Platform and Plugin Updates: PyCharm is ready to update. (today 6:27)
17:109 n/a UTF-8

```
#    }
#
#}
from elasticsearch import Elasticsearch
esclient = Elasticsearch(['localhost:9200'])
response = esclient.search(
index='logstash-*',
body={
    "query": {
        "match": {
            "URL": "s0.2mdn.net"
```

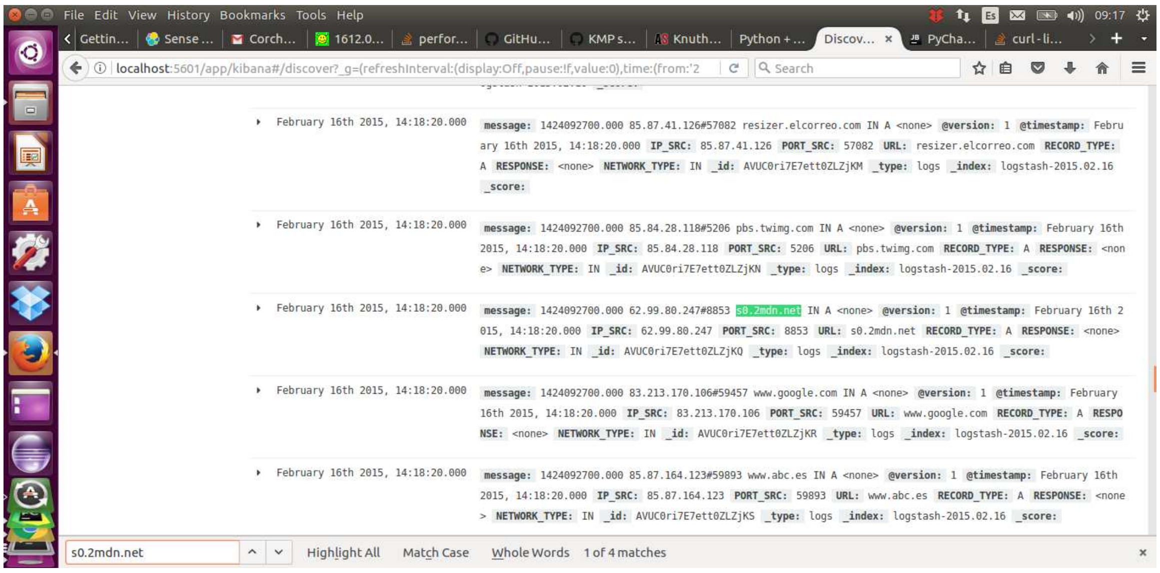
```
/usr/bin/python2.7 /home/oscar4/PycharmProjects/pythonES1/ES2python6.py
s0.2mdn.net
URL maliciosa from MyQLS: ('s0.2mdn.net',)
{
  "name" : "Eel",
  "cluster_name" : "elasticsearch",
  "version" : {
    "number" : "2.3.3",
    "build_hash" : "218bdf10790eef486ff2c41a3df5cfa32dadcfde",
    "build_timestamp" : "2016-05-17T15:40:04Z",
    "build_snapshot" : false,
    "lucene_version" : "5.5.0"
  },
  "tagline" : "You Know, for Search"
}

{u'hits': {u'hits': [{u'_score': 7.8843913, u'_type': u'logs', u'_id': u'AVUCq22nE7ett0ZLTrR-', u'_source': {u'URL': u's0.2mdn.net', u'@timestamp': u'2015-02-16T13:16:

Process finished with exit code 0
```

Figure 4.5: Finding a "suspicious" URL in use in several different smart devices in the DNS-service network logs using the first Python-language script.

Moreover, two programs written in Python language are used for this subsection. The first Python script extracts one suspicious URL upon the time from the MySQL database with the App traces, and then it connects with *ES* to look up through the whole indexed DNS Server records within it. For instance, if we search for the particular URL, **s0.2mdn.net**. The Python program obtained automatically (see, Figure 4.5) this target URL from app traces stored in the MySQL database in the service cloud. In Figure 4.6, it is shown one of the DNS logs from the mobile operator and obtained after the processing (indexing) with the tool *Logstash* and stored in *ES*. Also, in the Figure 4.6, we can appreciate the several fields that can be utilized to look up for precise information, in our case we search for the field tagged "URL". In the case of **s0.2mdn.net**, as a matter of fact, this URL is actually an adware accordingly to [96].

In Table 4.1, we can see the possibility of finding more malware not only in the smartphone under examination, but in other smart devices that are concurrently using the same malicious URLs and are also being detected in the DNS network-service traffic so we can do a decisions correlation.

The second Python script allows us to do pattern matching in strings using a well-known algorithm (or KMP algorithm) [116]. The pattern matching is done to compare the *Elasticsearch* processing against the KMP algorithm. In other words, the URLs stored in the MySQL database of the cloud service are also read with the second Python script. After that, we run the KMP algorithm, to conduct a pattern matching in strings, searching directly inside the DNS Server log in *ES* to look for the URLs under examination. Comparison of the searching time on malicious URLs inside the indexed DNS log in *ES* using the first Python script are faster by approximately five times in average in 10000 trials, versus the searching time of using the second Python script.



File Edit View History Bookmarks Tools Help
09:17
Gettin… Sense … Corch… 1612.0… perfor… GitHu… KMP s… Knuth… Python + … Discov… PyCha… curl-li…
localhost:5601/app/kibana#/discover?_g=(refreshInterval:(display:Off,pause:!f,value:0),time:(from:'2
Search
s0.2mdn.net
Highlight All
Match Case
Whole Words
1 of 4 matches

```
February 16th 2015, 14:18:20.000   message: 1424092700.000 85.87.41.126#57082 resizer.elcorreo.com IN A <none> @version: 1 @timestamp: Febru
                                   ary 16th 2015, 14:18:20.000 IP_SRC: 85.87.41.126 PORT_SRC: 57082 URL: resizer.elcorreo.com RECORD_TYPE:
                                   A RESPONSE: <none> NETWORK_TYPE: IN _id: AVUC0ri7E7ett0ZLZjKM _type: logs _index: logstash-2015.02.16
                                   _score:

February 16th 2015, 14:18:20.000   message: 1424092700.000 85.84.28.118#5206 pbs.twimg.com IN A <none> @version: 1 @timestamp: February 16th
                                   2015, 14:18:20.000 IP_SRC: 85.84.28.118 PORT_SRC: 5206 URL: pbs.twimg.com RECORD_TYPE: A RESPONSE: <non
                                   e> NETWORK_TYPE: IN _id: AVUC0ri7E7ett0ZLZjKN _type: logs _index: logstash-2015.02.16 _score:

February 16th 2015, 14:18:20.000   message: 1424092700.000 62.99.80.247#8853 s0.2mdn.net IN A <none> @version: 1 @timestamp: February 16th 2
                                   015, 14:18:20.000 IP_SRC: 62.99.80.247 PORT_SRC: 8853 URL: s0.2mdn.net RECORD_TYPE: A RESPONSE: <none>
                                   NETWORK_TYPE: IN _id: AVUC0ri7E7ett0ZLZjKQ _type: logs _index: logstash-2015.02.16 _score:

February 16th 2015, 14:18:20.000   message: 1424092700.000 83.213.170.106#59457 www.google.com IN A <none> @version: 1 @timestamp: February
                                   16th 2015, 14:18:20.000 IP_SRC: 83.213.170.106 PORT_SRC: 59457 URL: www.google.com RECORD_TYPE: A RESPO
                                   NSE: <none> NETWORK_TYPE: IN _id: AVUC0ri7E7ett0ZLZjKR _type: logs _index: logstash-2015.02.16 _score:

February 16th 2015, 14:18:20.000   message: 1424092700.000 85.87.164.123#59893 www.abc.es IN A <none> @version: 1 @timestamp: February 16th
                                   2015, 14:18:20.000 IP_SRC: 85.87.164.123 PORT_SRC: 59893 URL: www.abc.es RECORD_TYPE: A RESPONSE: <none
                                   > NETWORK_TYPE: IN _id: AVUC0ri7E7ett0ZLZjKS _type: logs _index: logstash-2015.02.16 _score:
```

Figure 4.6: Indexed DNS log after being processed by *Logstash* tool and stored in the *ES*, and it is shown using the tool for searching and data visualization called *Kibana*. The queried URL is **s0.2mdn.net**.

Table 4.1: MALICIOUS URLs AFFECTING SEVERAL SMART DEVICES

Malicious URLs from smartphones	No. of times in one of the DNS Server logs from a mobile operator
dlinkddns.com	2221
xxcamd.com	1806
alog.umeng.com	1630
thepiratebay.org	1024
servegame.com	288
pm-m.d.chango.com	30

4.3.2 Experimental Results with ML

In order to improve the proposed framework, we add ML techniques. The assessment of the SVM and RF classifiers are done by using 10-folds cross-validation. In this case, the dataset is divided into 10 parts or sets, 9 used to training and one for teasing process. Python with the scikit-learn can provides us with the accuracy of the classifier and gives information about the actual and predicted classifications done by the system. The so-called confusion matrix (Figure 4.7) is composed of following terms includes: TP is also known as hit, False alarm or Type I error (FP), True Negative (TN), False Negative (FN) or Type II error. Then we use the following equations below to compute the main metrics of precision or positive predictive value (PPV), recall or True Positive Rate (TPR), specificity or True Negative Rate (TNR), and F1-Score.

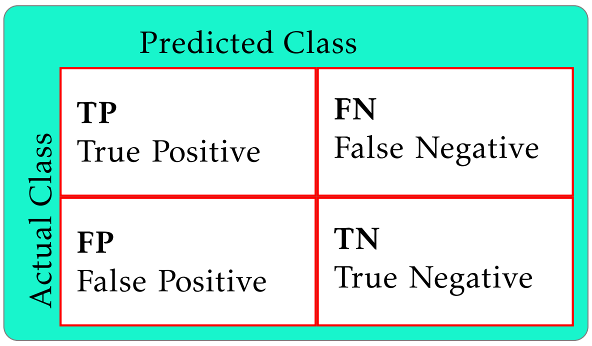


Figure 4.7: Confusion matrix: TP as true positive, TN as correct rejection, FP as false positive and FN miss. P_{PV} stands for Positive Predictive Value (also Precision), T_{PR} True Positive Rate (Sensitivity) T_{NR} , True Negative Rate (Specificity)and Negative Predictive Value (False Positive Rate)

The definition in machine learning of these parameters (Figure 4.8) depends upon the different entries of what is known as a confusion matrix or error matrix. That matrix is a specific table layout that allows visualization of the performance of an algorithm and it is an array with two rows and two columns (see Figure 4.7) that reports the number of false positives, false negatives, true positives, and true negatives regarding a classification task where the goal is to predict an outcome from a process.

Precision $PPV = \frac{T_P}{T_P + F_P}$	**Sensitivity** $T_{PR} = \frac{T_P}{T_P + F_N}$	**Neg.Pred. Value** $NPV = \frac{T_N}{T_N + F_N}$
Specificity $T_{NR} = \frac{T_N}{T_N + F_P}$	**Fals. Pos. Rate** $FPR = \frac{F_P}{F_P + T_N}$	**Accuracy** $ACC = \frac{T_P + T_N}{P + N}$

Figure 4.8: The different parameters used to describe error performance. P_{PV} stands for Positive Predictive Value (same as Precision), True Positive Rate T_{PR} (Sensitivity), True Negative Rate T_{NR} (same as Specificity), Negative Predictive Value N_{PV} (False Positive Rate)

Receiver operating characteristic (ROC) curves are also frequently used in algorithm performance evaluation. These are 2D plot where sensitivity is plotted against 1-specificity (False Positive Rate). When we require a unique value the F1-score can be used as a single measure of performance of the test. The F1-Score is the harmonic mean of precision (also called PPV or Positive Predictive Value) and recall (Sensitivity, TPR) and it is defined as follows:

$$F1 - Score = 2 * \frac{PPV * TPR}{PPV + TPR} = 2 * \frac{Precision * Recall}{Precision + Recall}. \tag{4.1}$$

In the experiment with the SVM algorithm, the TP (equivalent with hit) of malicious domains detected resulted in 3666 out of 5000 predicted malware domains, which leaves behind an FP (equivalent with false alarm) of 1334 (TP of the malicious domains is 0.7332). For the benign domains, the TP outcome was 3980 out of 4,500 predicted legitimate domains, with a FP of 520 (TP of the legitimate domains is 0.8844).

The Table below (The accuracy of the SVM Classifier) shows the results received from the SVM classifier. The highest classification detection rate was 87.58% percent. Out of a total of 9,500 malicious and legitimate domains: 7646 domains were classified correctly, and 1854 were wrong classified. The precision is the fraction of retrieved instances that are relevant, whereas Recall is the fraction of relevant instances that are retrieved. Precision for malicious domains=

3666/(3666+ 520) = 0.8757, and the Precision for benign domains=3980/(3980+ 1334) = 0.7489.

The accuracy of SVM Classifier			
	Malicious	Benign	Weighted Avg
TP Rate	0.7332	0.8844	0.8048
FP Rate	0.1155	0.2668	0.1156
Precision	0.8757	0.7489	0.8758
Recall	0.7050	0.8572	0.7332
F1-Score	0.7982	0.7982	0.7982

Regarding the experiments with Random Forest (FR) algorithm, we obtained the following results shown in Table 4.2:

Table 4.2: The experimental results for the Random Forest algorithm

TP Rate	0.885
FP Rate	0.130
Precision	0.885
Recall	1024
F1-Score	0.885

Now, using one of the DNS server logs from our main campus at UNI combined with the framework with ML (*SIMPLEDNS*), we detected the following Top 7 malicious URLs, namely as it is show in Table 4.3 below:

Table 4.3: MALICIOUS URLs AFFECTING SEVERAL SMART DEVICES

Malicious URLs from smartphone traffic	**No. of times in the DNS logs**
http://market.moboplay.com/softs.ashx	478
http://y-bt.in/	396
http://ya.ru	336
http://search.gongfu-android.com	284
http://222.186.37.93:9000/Application/reportStateC.do	138
t-mysqlnet.com	130
http://cecilia-gilbert.com/	25

4.4 Limitations

It is very-well known, there is no prefect detection system without limitations. In this chapter, we only consider URL/DNS traffic, however current applications and malware as well are using DNS tunneling techniques and HTTP traffic so, taking into consideration the current communication landscape this proposal will be only covering partially the current malware. In addition to this, malware using certificate pinning will be totally able to evade our system. Also, working with sandboxes in DA instead of real Android phone could be a drawback since malware can detect the use of emulators as is discussed in [117]. For instance, most of the current mobile malware, specially bankers make use of geo-location techniques in order to prevent it being execute in sandboxes, so this is one of the vectors the system must consider. In order to develop a real-time system for malware detection as a Framework for Mining Massive Malware Data on Mobile Networks and Devices, it is necessary to complement *SIMPLEDNS* with a big data platform since *Elasticsearch* is more suitable and limited for full-text analysis. As a matter of fact, several graph approaches have been proposed in the literature [16, 118] for discovering malicious domains through DNS Data graph analysis where *Elastic stack* is not enough in resources and computational power to handle this new direction. Therefore, these issues will need further research work.

4.5 Conclusions of the Chapter

In this chapter, we propose a collaborative framework (*SIMPLEDNS*) for Android MD that allows finding events correlation among common malicious URLs from the App traces in the smart device and the DNS-service network logs from the mobile operator. This can be used to pinpoint the malware attacks in several unmonitored smartphones in the wireless cellular system. This platform provides a visualization component using the tool dubbed *Kibana* from the *Elastic Stack*, in particular, the malicious URLs corresponding to malware behaviors are highlighted. Our infrastructure is composed of several components namely: the App that collects the network traffic of the application under examination, Python-language scripts that allow processing of the URLs taking from the App traces and the DNS records at the server that supports the search and analytics cluster (*Elastic Stack*) and the ML module. So, any Android application (up to 19 API level) can be monitored without rooting the phone or changing its firmware. Our results with the proposed DNS framework using the RF algorithm shows, regarding the F1-Score, a better performance than the work in [99] with J.48 classifier (0.885 versus 0.775). Further improvements on the visualization quality and the user interface are possible, but the design and implementation of our platform demonstrated to be promising.

Chapter 5

Conclusions and Future Work

This chapter presents the conclusions of this dissertation. We first summarize the main contributions. Thereafter, we identify a number of challenging open issues/future research lines that need further research work.

5.1 Conclusions

This thesis has dealt mainly with the design and implementation of a lightweight framework for detecting mobile smart malware based on a dynamic DNS monitoring approach. This is a potential technology to improve cybersecurity, specifically in mobile platforms that it consists of efficiently monitoring mobile communications for the early identification of new attacks and to limit their impact. The selection of this technology has been proposed as an appealing solution to tackle mobile threats. Concrete our framework will take special care to minimize the impact that the use of mobile devices will have on their performance when they are acting as information collectors (i.e., some distributed sensors) for the monitoring system. The information collected from the devices along with the data log collection in the networks of the operator will be combined for the monitoring, detection, characterization, and mitigation of mobile threats as well as to create an early warning system for the operators. We first conducted a thorough literature survey where we analyzed the field of mobile security, in particular, regarding Android malware analysis techniques. We identified some gaps in current research and some headroom for future betterments/enhancements based on dynamic analysis. So, we investigated existing Android malware detection algorithms and methods, in terms of computing cost and detection strategy. We identify that the abuse of the DNS system is, in most of the cases, the first step to launch a malicious attack to the smart device. Thus we must take into account this fact to detect mobile malware.

Next, we proposed an infrastructure named *AppShaper* for monitoring the Android applications in a platform-independent manner. The infrastructure consists of two parts: one in the smartphone to collect the different partial traces issued by the monitored Apps by means of another Android application dubbed the Sink; and the server side to collect the traces from the smartphones and prepare the applications (i.e., inserting the hooks and the communication module for sending the logs to the Sink). We have evaluated the performance of the Sink upon receiving a high quantity of partial traces. It turns out that the Sink is capable of generating traces for storage in a local database with reasonable CPU usage (not exceeding 28% even with an unrealistically high load). As for the monitored application, we find that its CPU utilization, due to insertion of each probe, is negligible for the responsiveness (in the order of milliseconds). In other words, our approach includes platform-independent application instrumentation, introducing hooks in order to trace restricted API functions used at runtime of the application. These function calls were collected at a central server where the App behavior filtering and visualization take place (through the usage of graphs and dendroid diagrams). In this way, detection and visualization of Android malware behavior were achieved through a rule-based system based mainly on API calls without compromising the performance of the mobile devices involved. This result can help Android malware analysts to inspect visually what the application under study does, easily identifying such malicious functions.

Later, we designed and implemented a dynamic DNS Android Sniffer termed *Network Sentinel* based on VPN approach, which enabled us to capture and store in near-real time the DNS queries request done by Apps in a smart device. This will be the client side of the second Framework for mining malware data on mobile networks and devices by DNS traffic termed *SIMPLEDNS*. By doing this, it was possible to design and develop a system capable of extracting the DNS queries requested by the Android applications (i.e., the work done here is focused on intercepting the domains requested by application under test), in an efficient way regarding the resources of the smart device. The App *Network Sentinel* is publicly available at Google Store and, it has capabilities to classify URLs based on two well-known public Reputation Blacklist services available on Internet (this will be our first line of defense in our framework *SIMPLEDNS*). If one URL is found suspected or malicious, it is then stored in its internal database. All the captured URLs are sent to a remote server side for further processing, see Chapter 4.

Finally, we modified the framework in Chapter 2 to add a convenient, suitable server side (the Cloud side) where we utilized the *Elastic Stack* (*Elasticsearch*, *Logstash*, and *Kibana*) to provide a distributed, RESTful search and analytics engine capable of solving a growing number of use cases. As the heart of the *Elastic Stack*, *Elasticsearch* centrally stores our gathered data so that we can conduct string pattern matching. As a second line of defense on the remote server side, we also trained and developed a machine learning (ML) module based on lexical analysis and heuristic rules to obtain a classifier system of the URLs, which have been sent previously by the App *Network Sentinel* on the smart device. The execution of malware implies, in most of the times, the access of the same malicious domains on Internet, which is a reason why the capture of this type of consultations could help us to detect infected devices without needing to monitor them directly. In other words, we identified within the DNS records from the DNS Servers of the network those requests that utilize the device running the Android application under test. And we can then map the extracted DNS records into the server side to classify each one of them as a benign or malicious domain (the ML algorithm does this). Thus, we developed a framework that automates the recent analysis using our client(s) App(s) for smart devices developed or enhanced in this research. In summary, in this thesis, we explored, designed, and developed techniques that can be used to detect the malicious mobile behavior of medium-size collections (hundreds to thousands of traces) of heterogeneous sources.

5.2 Future Work

In this section, we outline future research directions that can lead to additional contributions to the field of mobile security. This thesis, though limited in scope, offers development opportunities that deserve the attention of the scientific community for further advances in the field. We now list the aforementioned opportunities arranged by topic.

Extending the framework *AppShaper* (Chapter 2). This monitoring platform could be extended by logging the selected parameters of hooked functions too. However, before doing that it is needed to address the privacy issues since we will be able to log private information.

Extending the framework *SIMPLEDNS* into a Big Data Testbed (Chapter 4). Instead of using the *Elastic Stack* as in this dissertation is done, this framework can be redesigned and implemented as a Big Data testbed through the usage of the *Apache Spark* platform, which is an open-source distributed general-purpose cluster-computing framework, allowing data parallelism and fault tolerance; using this framework will allow us to add graph analysis to the malware detection. On the other hand, *Elasticsearch*, in a nutshell, is a search engine. It is great at getting documents, understanding their language (word stemming, cleaning stop words, etc.), and storing them in a way that will allow a very fast fetching. However, it will be fast only for fetching the first 100s-1000s of documents. When querying for 10000 results - *Elasticsearch* will be relatively slow. In short, the huge number of malware samples and big size of collected data for malware detection cause a big data problem, which challenges the detection and forces it to be much efficient to handle big data. The speed of malware growth has never slowed down. The sample database is becoming enormous. So, finding a way to dramatically and efficiently reduce the sample space and effectively detect malware is urgent. To some possible extent, distributed detection systems and cloud-based solutions can make this problem easier to solve. Besides, data mining and fusion methods and some other strategies used in big data processing can also be applied in solving the big data issues in this research field. Providing more intelligence to the smart device. The trained machine learning model based on lexical features developed on the server or cloud side in Chapter 4 of this thesis, it might be transferred to the smart device enhancing its classification capabilities. Since this model mentioned above is a light-weight method. Now the mobile phone will have two lines of defenses, one through the usage of blacklists and the second one by means of the trained machine learning model incorporated into de smartphone, either option can classify a URL as malicious, implying that the server side or cloud service will be informed about this behavior, and it will take the necessary actions to detect the same malware in another mobile device (which are not being currently monitored) active in the DNS logs from the DNS servers.

Addressing Protection Privacy Issues to the frameworks. In the literature, we found many cloud-based methods proposed for mobile malware detection due to the constrained resources of smart devices. This is also valid for the aforementioned methods suggested in this thesis work, which carried out feature analysis and detection of unknown software on a server or a cloud, that could result in privacy concerns. The collected data about mobile users need to be uploaded to the cloud to be processed. This could intrude on user privacy since the cloud cannot be fully trusted. But, none of the existing work provided privacy protection during the process of mobile malware detection. This is an important private data leakage issue that urges in future research efforts to be addressed.

Enhancing the proposed framework *SIMPLEDNS* through Content-based Features. Content-based features are those obtained upon downloading the entire web-page, so it must be done at the server or cloud side. As compared to URL-based features, these are heavy-weight traits, as a lot of information needs to be extracted, and at the same time, safety concerns may arise. The content-based features of a web-page can be drawn primarily from its HTML content, and the usage of JavaScript. Usually, this approach is combined with machine learning algorithms. In the sequel, a general processing framework for malicious URL detection using machine learning is suggested in [115], including the formulation of the binary classification problem as a convex optimization process, see([115], page 5, equation 1).

References

[1] Asier Aduriz Saiz, "Simple: Security in mobile platforms with even analysis.," july 2014, Mondragon Unibertsitatae.

[2] THe Guardain, "How the smartphone is killing the pc. retrieve april 30, 2017 from https://www.theguardian.com/technology/2011/jun/05/smartphones-killing-pc.," Available Online, 2017.

[3] Ping Yan and Zheng Yan, "A survey on dynamic mobile malware detection," *Software Quality Journal*, pp. 1–29, 2017.

[4] IDC, ""smartphone shipments os market share, 2017 q1," http://www.idc.com/prodserv/smartphone-os-market-share.jsp.," Available Online, 2017.

[5] Statista, "Number of available applications in the google play store from december 2009 to march 2017. retrieve april 18, 2017 from https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store/.," Available Online, 2017.

[6] G. DATA, "8,400 new android malware samples every day. retrieve april 18, 2017 from https://www.gdatasoftware.com/blog/2017/ 04/29712-8-400-new-android-malware-samples-every-day," Available Online.

[7] F-secure, "F-secure-threat-report-state of cyber security 2017. retrieve july 30, 2017 from http://www.f-secure.com.," Available Online, 2017.

[8] eMarketer, "Mobile phone users and penetration worldwide, 2015-2020. retrieve july 30, 2017 from https://www.emarketer.com/chart/mobile-phone-users-penetration-worldwide-2015-2020-billions-of-population-change/196278," Available Online, 2017.

[9] SophosLabs 2018, "Sophoslabs 2018 malware forecast. retrieve september 15, 2018 from https://www.sophos.com/en-us/en-us/medialibrary/pdfs/technical-papers/malware-forecast-2018.pdf?la=en," Available Online, 2017.

[10] Zhiqiang Wu, Xin Chen, and Scott Uk-Jin Lee, "Identifying latent android malware from applications description using lstm," .

[11] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 961–987, 2014.

[12] Kimberly Tam, Salahuddin J Khan, Aristide Fattori, and Lorenzo Cavallaro, "Copperdroid: Automatic reconstruction of android malware behaviors.," in *NDSS*, 2015.

[13] Chanwoo Bae and Seungwon Shin, "A collaborative approach on host and network level android malware detection," *Security and Communication Networks*, vol. 9, no. 18, pp. 5639–5650, 2016.

[14] 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*. ACM, 2014, pp. 1105–1116.

[15] Marián Kühnel and Ulrike Meyer, "Applying highly space efficient blacklisting to mobile malware," *Logic Journal of the IGPL*, vol. 24, no. 6, pp. 971–981, 2016.

[16] Matija Stevanovic, Jens Myrup Pedersen, Alessandro DAlconzo, and Stefan Ruehrup, "A method for identifying compromised clients based on dns traffic analysis," *International Journal of Information Security*, pp. 1–18, 2016.

[17] Nguyen Tan Cam and Nguyen Cam Hong Phuoc, "Nesedroid android malware detection based on network traffic and sensitive resource accessing," in *Proceedings of the International Conference on Data Engineering and Communication Technology*. Springer, 2017, pp. 19–30.

[18] Darell JJ Tan, Tong-Wei Chua, Vrizlynn LL Thing, et al., “Securing android: A survey, taxonomy, and challenges,” *ACM Computing Surveys (CSUR)*, vol. 47, no. 4, pp. 58, 2015.

[19] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro, “The evolution of android malware and android analysis techniques,” *ACM Computing Surveys (CSUR)*, vol. 49, no. 4, pp. 76, 2017.

[20] William Enck, Machigar Ongtang, and Patrick McDaniel, “Understanding android security,” *IEEE security & privacy*, vol. 7, no. 1, pp. 50–57, 2009.

[21] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer, “Google android: A comprehensive security assessment,” *IEEE Security & Privacy*, vol. 8, no. 2, pp. 35–44, 2010.

[22] Aubrey-Derrick Schmidt, *Detection of smartphone malware*, Phd thesis, Technical University of Berlin, 2011.

[23] Mohsen Damshenas, Ali Dehghantanha, and Ramlan Mahmoud, “A survey on malware propagation, analysis, and detection,” *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, vol. 2, no. 4, pp. 10–29, 2013.

[24] Carlos A Castillo et al., “Android malware past, present, and future,” *White Paper of McAfee Mobile Security Working Group*, vol. 1, pp. 16, 2011.

[25] Mariantonietta La Polla, Fabio Martinelli, and Daniele Sgandurra, “A survey on security for mobile devices,” *IEEE communications surveys & tutorials*, vol. 15, no. 1, pp. 446–471, 2013.

[26] 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,” *IEEE communications surveys & tutorials*, vol. 17, no. 2, pp. 998–1022, 2014.

[27] 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 Universität Darmstadt, Apr. 2011.

[28] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck, "Structural detection of android malware using embedded call graphs," in *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. ACM, 2013, pp. 45–54.

[29] Daniel Arp, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, Konrad Rieck, and CERT Siemens, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proceedings of the Annual Symposium on Network and Distributed System Security (NDSS)*, 2014.

[30] Xuxian Jiang and Yajin Zhou, "A survey of android malware," in *Android Malware*, pp. 3–20. Springer, 2013.

[31] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou, "Deep ground truth analysis of current android malware," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2017, pp. 252–276.

[32] Bahman Rashidi and Carol J Fung, "A survey of android security threats and defenses.," *JoWUA*, vol. 6, no. 3, pp. 3–35, 2015.

[33] Alireza Sadeghi, Hamid Bagheri, Joshua Garcia, and Sam Malek, "A taxonomy and qualitative comparison of program analysis techniques for security assessment of android software," *IEEE Transactions on Software Engineering*, vol. 43, no. 6, pp. 492–530, 2017.

[34] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Computing Surveys (CSUR)*, vol. 44, no. 2, pp. 6, 2012.

[35] Oscar Somarriba and Henry Jaentschke, "Dynamic android malware detection: A survey," in *IEEE LATINCOMM 2017*. IEEE, 2017.

[36] Yajin Zhou and Xuxian Jiang, "Dissecting android malware: Characterization and evolution," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 95–109.

[37] Vanja Svajcer, "Sophos mobile security threat report," in *Launched at Mobile World Congress*, 2014.

[38] Michael Sikorski and Andrew Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*, no starch press, 2012.

[39] 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*, vol. 41, no. 4, pp. 1104–1117, 2014.

[40] 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*. ACM, 2011, pp. 15–26.

[41] Kanubhai Patel and Bharat Buddhadev, "Predictive rule discovery for network intrusion detection," in *Intelligent Distributed Computing*, pp. 287–298. Springer, 2015.

[42] Guihua Shan, Yang Wang, Maojin Xie, Haopu Lv, and Xuebin Chi, "Visual detection of anomalies in dns query log data," in *Visualization Symposium (PacificVis), 2014 IEEE Pacific*. IEEE, 2014, pp. 258–261.

[43] Michael Dooley and Timothy Rooney, *DNS Security Management*, John Wiley & Sons, 2017.

[44] Manos Antonakakis, Roberto Perdisci, Wenke Lee, Nikolaos Vasiloglou, and David Dagon, "Detecting malware domains at the upper dns hierarchy.," in *USENIX security symposium*, 2011, vol. 11, pp. 1–16.

[45] Leyla Bilge, Engin Kirda, Christopher Kruegel, and Marco Balduzzi, "Exposure: Finding malicious domains using passive dns analysis.," in *Ndss*, 2011.

[46] Duane Wessels, Marina Fomenkov, Nevil Brownlee, et al., "Measurements and laboratory simulations of the upper dns hierarchy," in *International Workshop on Passive and Active Network Measurement*. Springer, 2004, pp. 147–157.

[47] David Dagon, Cliff Changchun Zou, and Wenke Lee, "Modeling botnet propagation using time zones.," in *NDSS*, 2006, vol. 6, pp. 2–13.

[48] Linh Vu Hong, "Dns traffic analysis for network-based malware detection," M.S. thesis, 2012.

[49] Shuang Hao, Nick Feamster, and Ramakant Pandrangi, "An internet-wide view into dns lookup patterns," *School of Computer Science, Georgia Tech, Tech. Rep*, 2010.

[50] Charles Lever, Manos Antonakakis, Bradley Reaves, Patrick Traynor, and Wenke Lee, "The core of the matter: Analyzing malicious traffic in cellular carriers.," in *NDSS*, 2013.

[51] Shree Garg, Sateesh K Peddoju, and Anil K Sarje, "Network-based detection of android malicious apps," *International Journal of Information Security*, pp. 1–16, 2016.

[52] Nigel Cross, "Science and design methodology: a review," *Research in engineering design*, vol. 5, no. 2, pp. 63–69, 1993.

[53] Dawn G Gregg, Uday R Kulkarni, and Ajay S Vinzé, "Understanding the philosophical underpinnings of software engineering research in information systems," *Information Systems Frontiers*, vol. 3, no. 2, pp. 169–183, 2001.

[54] R Hevner Von Alan, Salvatore T March, Jinsoo Park, and Sudha Ram, "Design science in information systems research," *MIS quarterly*, vol. 28, no. 1, pp. 75–105, 2004.

[55] Juhani Iivari, "A paradigmatic analysis of information systems as a design science," *Scandinavian journal of information systems*, vol. 19, no. 2, pp. 5, 2007.

[56] Ken Peffers, Tuure Tuunanen, Charles E Gengler, Matti Rossi, Wendy Hui, Ville Virtanen, and Johanna Bragge, "The design science research process: a model for producing and presenting information systems research," in *Proceedings of the first international conference on design science research in information systems and technology (DESRIST 2006)*. sn, 2006, pp. 83–106.

[57] Oscar Somarriba, "Detecting blacklisted urls from unmodified and non-rooted android devices," in *Central America and Panama Convention (CONCAPAN XXXVll), 2017 IEEE 37th*. IEEE, 2017, pp. 1–6.

[58] Oscar Somarriba, Luis Carlos Perez Ramos, Urko Zurutuza, and Roberto Uribeetxeberria, "Dynamic dns request monitoring of android applications via networking," in *2018 IEEE 38th Central America and Panama Convention (CONCAPAN XXXVIII)*. IEEE, 2018, pp. 1–6.

[59] Oscar Somarriba, Ignacio Arenaza Nuño, Roberto Uribeetxeberria, and Urko Zurutuza, "Análisis visual del comportamiento de aplicaciones para android," 2014.

[60] Oscar Somarriba, Urko Zurutuza, Roberto Uribeetxeberria, Laurent Delosières, and Simin Nadjm-Tehrani, "Detection and visualization of android malware behavior," *Journal of Electrical and Computer Engineering*, vol. 2016, 2016.

[61] Oscar Somarriba and Urko Zurutuza, "A collaborative framework for android malware detection using dns and dynamic analysis," in *IEEE Central American and Panama Convention (CONCAPAN XXXVII). Nov 2017*. IEEE, 2017.

[62] 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*. ACM, 2012, pp. 137–148.

[63] 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 & communications security*. ACM, 2013, pp. 611–622.

[64] Android.com, "Android system permissions," Available online, 2014.

[65] R Winsniewski, "Android–apktool: A tool for reverse engineering android apk files," 2012.

[66] M. Karami, M. Elsabagh, P. Najafiborazjani, and A. Stavrou, "Behavioral analysis of android applications using automated instrumentation," in *Software Security and Reliability-Companion (SERE-C), IEEE 7th International Conference on*, June 2013, pp. 182–187.

[67] 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*. ACM, 2012, pp. 217–228.

[68] 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*. ACM, 2012, pp. 3–14.

[69] A Shabtai, L Tenenboim-Chekina, D Mimran, L Rokach, B Shapira, and Y Elovici, "Mobile malware detection through analysis of deviations in application network behavior," *Computers & Security*, vol. 43, pp. 1–18, 2014.

[70] Quan Qian, Jing Cai, Mengbo Xie, and Rui Zhang, "Malicious behavior analysis for android applications," *International Journal of Network Security*, vol. 18, no. 1, pp. 182–192, 2016.

[71] Microsoft, "Web services," Available Online, 2014.

[72] Roy Thomas Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, Phd thesis, University of California, Irvine, 2000.

[73] Steven Arzt, Siegfried Rasthofer, and Eric Bodden, "Instrumenting android and java applications as easy as abc," in *Runtime Verification*. Springer, 2013, pp. 364–381.

[74] Seung-Hyun Seo, Aditi Gupta, Asmaa Mohamed Sallam, Elisa Bertino, and Kangbin Yim, "Detecting mobile malware threats to homeland security through static analysis," *Journal of Network and Computer Applications*, vol. 38, pp. 43–53, 2014.

[75] Kanubhai Patel and Bharat Buddadev, "Detection and mitigation of android malware through hybrid approach," in *Security in Computing and Communications*, pp. 455–463. Springer, 2015.

[76] Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, Guillermo Suarez-Tangil, and Steven Furnell, "Androdialysis: Analysis of android intent effectiveness in malware detection," *computers & security*, vol. 65, pp. 121–134, 2017.

[77] Parag H Rughani, "Detecting blacklisted ip access from android phone," *Indian Journal of Science and Technology*, vol. 9, no. 48, 2016.

[78] Usha Banerjee, Ashutosh Vashishtha, and Mukul Saxena, "Evaluation of the capabilities of wireshark as a tool for intrusion detection," *International Journal of computer applications*, vol. 6, no. 7, 2010.

[79] Jae Kyu Lee, "Research framework for ais grand vision of the bright ict initiative," *MIS Quarterly*, vol. 39, no. 2, 2015.

[80] Shree Garg, Sateesh K Peddoju, and Anil K Sarje, "Network-based detection of android malicious apps," *International Journal of Information Security*, pp. 1–16, 2016.

[81] Taosoftware, "tpacketcapture is the software that can capture communication packets on non-rooted device," *Online: http://www.taosoftware.co.jp/en/android/packetcapture/*, 2015.

[82] Nachiket Trivedi and Manik Lal Das, "Maldetec: A non-root approach for dynamic malware detection in android," in *International Conference on Information Systems Security*. Springer, 2017, pp. 231–240.

[83] Daifur Abubakar Girei, Munam Ali Shah, and Muhammad Bilal Shahid, "An enhanced botnet detection technique for mobile devices using log analysis," in *Automation and Computing (ICAC), 2016 22nd International Conference on*. IEEE, 2016, pp. 450–455.

[84] Juanru Li, Wenbo Yang, Junliang Shu, Yuanyuan Zhang, and Dawu Gu, "Indroid: An automated online analysis framework for android applications," *Crisis Intervention Team (CIT)*, 2014.

[85] Wireshark, "Wireshark: A network protocol analyzer for unix and windows," .

[86] Van Jacobson, Craig Leres, and Steven McCanne, "Tcpdump public repository," *Web page at http://www. tcpdump. org*, 2003.

[87] Filippo Alberto Brandolini, *Hooking Java methods and native functions to enhance Android applications security*, Ph.D. thesis, University of Bologna, 2016.

[88] Marvin Wißfeld, *ArtHook: Callee-side Method Hook Injection on the New Android Runtime ART*, Ph.D. thesis, Saarland University, 2015.

[89] Michael Backes, Sven Bugiel, Oliver Schranz, Philipp von Styp-Rekowsky, and Sebastian Weisgerber, "Artist: The android runtime instrumentation and security toolkit," in *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*. IEEE, 2017, pp. 481–495.

[90] Lukas Dresel, Mykolai Protsenko, and Tilo Müller, "Artist: the android runtime instrumentation toolkit," in *Availability, Reliability and Security (ARES), 2016 11th International Conference on*. IEEE, 2016, pp. 107–116.

[91] Valerio Costamagna and Cong Zheng, "Artdroid: A virtual-method hooking framework on android art runtime.," in *IMPS@ ESSoS*, 2016, pp. 20–28.

[92] Yihang Song and Urs Hengartner, "Privacyguard: A vpn-based platform to detect information leakage on android devices," in *Proceedings of the 5th Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*. ACM, 2015, pp. 15–26.

[93] Web of Trust, "Web reputation ratings," 2016.

[94] Google, "Google safe browsing," 2017.

[95] C. Mulliner, "Adbi," 2016.

[96] Virus Total, "Virustotal-free online virus, malware and url scanner," 2012.

[97] Zhongmin Dai, Tong-Wei Chua, Dinesh Kumar Balakrishnan, Vrizlynn LL Thing, et al., "Chat-app decryption key extraction through information flow analysis," 2017.

[98] URLVoid, "Check if a website is malicious/scam or safe/legit," 2015.

[99] Khulood Al Messabi, Monther Aldwairi, Ayesha Al Yousif, Anoud Thoban, and Fatna Belqasmi, "Malware detection using dns records and domain name features," in *Proceedings of the 2nd International Conference on Future Networks and Distributed Systems*. ACM, 2018, p. 29.

[100] Yury Zhauniarovich, Issa Khalil, Ting Yu, and Marc Dacier, "A survey on malicious domains detection through dns data analysis," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 67, 2018.

[101] Jyoti Malik and Rishabh Kaushal, "Credroid: Android malware detection by network traffic analysis," in *Proceedings of the 1st ACM Workshop on Privacy-Aware Mobile Computing*. ACM, 2016, pp. 28–36.

[102] Yung-Tsung Hou, Yimeng Chang, Tsuhan Chen, Chi-Sung Laih, and Chia-Mei Chen, "Malicious web content detection by machine learning," *Expert Systems with Applications*, vol. 37, no. 1, pp. 55–60, 2010.

[103] Davide Canali, Marco Cova, Giovanni Vigna, and Christopher Kruegel, "Prophiler: a fast filter for the large-scale detection of malicious web pages," in *Proceedings of the 20th international conference on World wide web*. ACM, 2011, pp. 197–206.

[104] Annabella Astorino, A Chiarello, Manlio Gaudioso, and Antonio Piccolo, "Malicious url detection via spherical classification," *Neural Computing and Applications*, vol. 28, no. 1, pp. 699–705, 2017.

[105] Kamal Alieyan, Ammar ALmomani, Ahmad Manasrah, and Mohammed M Kadhum, "A survey of botnet detection based on dns," *Neural Computing and Applications*, vol. 28, no. 7, pp. 1541–1558, 2017.

[106] DNScrypt, "Dnscrypt is a protocol that authenticates communications between a dns client and a dns resolver," .

[107] Scarlett Gourley and Hitesh Tewari, "Blockchain backed dnssec," in *International Conference on Business Information Systems*. Springer, 2018, pp. 173–184.

[108] Florian Weimer, "Passive dns replication," in *FIRST conference on computer security incident*, 2005, p. 98.

[109] BV Elasticsearch, "Elasticsearch," 2016.

[110] Virus Total, "Virustotal-free online virus, malware and url scanner," *Online: https://www. virustotal. com/en*, 2012.

[111] Hongbo Han, Zhenxiang Chen, Qiben Yan, Lizhi Peng, and Lei Zhang, "A real-time android malware detection system based on network traffic analysis," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2015, pp. 504–516.

[112] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, 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," *ACM Transactions on Computer Systems (TOCS)*, vol. 32, no. 2, pp. 5, 2014.

[113] Andi Fitriah Abdul Kadir, Natalia Stakhanova, and Ali Akbar Ghorbani, "Android botnets: What urls are telling us," in *International Conference on Network and System Security*. Springer, 2015, pp. 78–91.

[114] Anh Le, Athina Markopoulou, and Michalis Faloutsos, "Phishdef: Url names say it all," in *2011 Proceedings IEEE INFOCOM*. IEEE, 2011, pp. 191–195.

[115] Doyen Sahoo, Chenghao Liu, and Steven CH Hoi, "Malicious url detection using machine learning: a survey," *arXiv preprint arXiv:1701.07179*, 2017.

[116] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt, "Fast pattern matching in strings," *SIAM journal on computing*, vol. 6, no. 2, pp. 323–350, 1977.

[117] Mohammed K Alzaylaee, Suleiman Y Yerima, and Sakir Sezer, "Emulator vs real phone: Android malware detection using machine learning," in *Proceedings of the 3rd ACM on International Workshop on Security and Privacy Analytics*. ACM, 2017, pp. 65–72.

[118] Issa Khalil, Ting Yu, and Bei Guan, "Discovering malicious domains through passive dns data graph analysis," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*. ACM, 2016, pp. 663–674.