Universidad Carlos III de Madrid

PhD Thesis

MINING STRUCTURAL AND BEHAVIORAL PATTERNS IN SMART MALWARE

Author:

D. Guillermo Suarez-Tangil

Supervisors:

Dr. D. Juan E. Tapiador

Dr. D. Pedro Peris-Lopez

Department of Computer Science
PhD in Computer Science and Technology

October 2014

TESIS DOCTORAL

MINING STRUCTURAL AND BEHAVIORAL PATTERNS IN SMART MALWARE

Autor: D. Guillermo Suárez de Tangil
Directores: Dr. D. Juan Estevez Tapiador, Dr. D. Pedro Peris Lopez

Firma del Tribunal Calificador:

	Nombre y Apellidos	Firma
Presidente:	Prof. D. Javier López	.....
Vocal:	Prof. D. N. Asokan	.....
Secretario:	Prof. D. Jesus García Herrero	.....

Calificación: .....

Leganés, ... de ..... de 2014.

To my parents and my girlfriend.

Nothing in life is to be feared, it is only to be understood.

— Marie Curie —

Malware is definitely not an exception.

Acknowledgments

Foremost, I would like to express my most sincere gratitude to my supervisors Prof. Juan E. Tapiador and Prof. Pedro Peris-Lopez for paving the way to knowledge during this long—but yet entertaining—journey. To Juan for his unconditional, patient and enthusiastic dedication. Even the most grandiloquent words I am capable of putting together are, by far, not enough to express my gratitude. Not only he has enlightened me with a new perspective to approach research in such a thriving area as ours, but also offered me valuable discussions about science, literature, life, etc. As somebody once said, "true mentoring is more than just answering occasional questions or providing *ad hoc* help. It is about an ongoing relationship of learning, dialogue, and challenge". Definitely, he has disinterestedly offered me all that far beyond his duties. I can only take careful notes, keep working hard and hope that one day I will be able to mentor somebody that way. To Pedro, for his encouraging support during these last years. His guidance helped me to deal with the daily burdens arising from a PhD study and helped me as well to keeping highly motivated.

Besides my supervisors, I would like to give special thanks to Prof. Arturo Ribagorda for embracing me within his research group and for proving me with this great opportunity. I would also like to thank all my colleagues: Benjamín, Anabel, Agustín, Chema, Lorena, Esther, Almudena, José, Israel, Andrea, etc., for the experiences shared together and their inestimable company throughout these years. Not to for-

get, special thanks to my dear mates Jorge and Sergio for their great friendship. I could not have imagined having better escorts than them, no matter where: researching, teaching, traveling, hiking, or *festivaling*. To Jorge for his generosity, for all those great moments experienced and for the ones yet to come. To Sergio for his passion and goodness, it has been always inspiring and has helped me to become a better person.

My sincere thanks also goes to Prof. Roberto Di Pietro and Prof. Mauro Conti for taking me in into their research groups and for opening me the doors of their family. Their dedication and passion has bestowed on me fresh ideas and new perspectives that have definitely helped me to improve the quality of this dissertation. To all my fellow researchers Flavio, Agustí, Pau, Stefano, etc. for making the most out of my research visits. I would also like to thank all those who have provided valuable feedback, with special mention to the members of the evaluation committee.

Of course, this Thesis would not have been possible without the support of my beloved family and dear friends. To my father, Nicolás, wherever he is, and to mother and siblings Teresa, Jacobo and Gonzalo. They all know how important their help has been. Of course, to all my friends for struggling for my mental health and the well-being of what was left of me after every deadline: Tito, Pablo, Manuela, Hugo, Yere, Paula, Nuria & Pablo, Carlos, Lucía, Sara, Ana, Elena, Gema, Noelia, etc. Thanks! Finally, to my dear love Anna for her devotion, unswerving patience and her firm understanding. She has been the greatest support anybody can provide. Only she knows the importance of the work provided in this Thesis, even if it may sound like rocket science to her. Thus, credit also goes to her.

Guillermo.

Abstract

Smart devices equipped with powerful sensing, computing and networking capabilities have proliferated lately, ranging from popular smartphones and tablets to Internet appliances, smart TVs, and others that will soon appear (e.g., watches, glasses, and clothes). One key feature of such devices is their ability to incorporate third-party apps from a variety of markets. This poses strong security and privacy issues to users and infrastructure operators, particularly through software of malicious (or dubious) nature that can easily get access to the services provided by the device and collect sensory data and personal information.

Malware in current smart devices—mostly smartphones and tablets—has rocketed in the last few years, supported by sophisticated techniques (e.g., advanced obfuscation and targeted infection and activation engines) purposely designed to overcome security architectures currently in use by such devices. This phenomenon is known as the proliferation of *smart malware*. Even though important advances have been made on malware analysis and detection in traditional personal computers during the last decades, adopting and adapting those techniques to smart devices is a challenging problem. For example, power consumption is one major constraint that makes unaffordable to run traditional detection engines on the device, while externalized (i.e., cloud-based) techniques raise many privacy concerns.

This Thesis examines the problem of smart malware in such devices, aiming at

designing and developing new approaches to assist security analysts and end users in the analysis of the security nature of apps. We first present a comprehensive analysis on how malware has evolved over the last years, as well as recent progress made to analyze and detect malware. Additionally, we compile a suit of the most cutting-edge open source tools, and we design a versatile and multipurpose research laboratory for smart malware analysis and detection.

Second, we propose a number of methods and techniques aiming at better analyzing smart malware in scenarios with a constant and large stream of apps that require security inspection. More precisely, we introduce Dendroid, an effective system based on text mining and information retrieval techniques. Dendroid uses static analysis to measures the similarity between malware samples, which is then used to automatically classify them into families with remarkably accuracy. Then, we present Alterdroid, a novel dynamic analysis technique for automatically detecting hidden or obfuscated malware functionality. Alterdroid introduces the notion of differential fault analysis for effectively mining obfuscated malware components distributed as parts of an app package.

Next, we present an evaluation of the power-consumption trade-offs among different strategies for off-loading, or not, certain security tasks to the cloud. We develop a system for testing several functional tasks and metering their power consumption called Meterdroid. Based on the results obtained in this analysis, we then propose a cloud-based system, called Targetdroid, that addresses the problem of detecting targeted malware by relying on stochastic models of usage and context events derived from real user traces. Based on these models, we build an efficient automatic testing system capable of triggering targeted malware.

Finally, based on the conclusions extracted from this Thesis, we propose a number of open research problems and future directions where there is room for research.

Resumen

Los dispositivos inteligentes se han posicionado en pocos años como aparatos altamente populares con grandes capacidades de cómputo, comunicación y sensorización. Entre ellos se encuentran dispositivos como los teléfonos móviles inteligentes (o smartphones), las televisiones inteligentes, o más recientemente, los relojes, las gafas y la ropa inteligente. Una característica clave de este tipo de dispositivos es su capacidad para incorporar aplicaciones de terceros desde una gran variedad de mercados. Esto plantea fuertes problemas de seguridad y privacidad para sus usuarios y para los operadores de infraestructuras, sobre todo a través de software de naturaleza maliciosa (o malware), el cual es capaz de acceder fácilmente a los servicios proporcionados por el dispositivo y recoger datos sensibles de los sensores e información personal.

En los últimos años se ha observado un incremento radical del malware atacando a estos dispositivos inteligentes—principalmente a *smartphones*—y apoyado por sofisticadas técnicas diseñadas para vencer los sistemas de seguridad implantados por los dispositivos. Este fenómeno ha dado pie a la proliferación de malware inteligente. Algunos ejemplos de estas técnicas inteligentes son el uso de métodos de ofuscación, de estrategias de infección dirigidas y de motores de activación basados en el contexto. A pesar de que en las últimos décadas se han realizado avances importantes en el análisis y la detección de malware en los ordenadores personales, adaptar y portar estas técnicas a los dispositivos inteligentes es un problema difícil de resolver.

En concreto, el consumo de energía es una de las principales limitaciones a las que están expuestos estos dispositivos. Dicha limitación hace inasequible el uso de motores tradicionales de detección. Por el contrario, el uso de estrategias de detección externalizadas (es decir, basadas en la nube) suponen una gran amenaza para la privacidad de sus usuarios.

Esta tesis analiza el problema del malware inteligente que adolece a estos dispositivos, con el objetivo de diseñar y desarrollar nuevos enfoques que permitan ayudar a los analistas de seguridad y los usuarios finales en la tarea de analizar aplicaciones. En primer lugar, se presenta un análisis exhaustivo sobre la evolución que el malware ha seguido en los últimos años, así como los avances más recientes enfocados a analizar *apps* y detectar malware. Además, integramos y extendemos las herramientas de código abierto más avanzadas utilizadas por la comunidad, y diseñamos un laboratorio que permite analizar malware inteligente de forma versátil y polivalente.

En segundo lugar, se proponen una serie de técnicas dirigida a mejorar el análisis de malware inteligente en escenarios dónde se requiere analizar importantes cantidad de muestras. En concreto, se propone Dendroid, un sistema basado en minería de textos que permite analizar conjuntos de apps de forma eficaz. Dendroid hace uso de análisis estático de código para extraer una medida de la similitud entre distintas las muestras de malware. Dicha distancia permitirá posteriormente clasificar cada muestra en su correspondiente familia de malware de forma automática y con gran precisión. Por otro lado, se propone una técnica de análisis dinámico de código, llamada Alterdroid, que permite detectar automáticamente funcionalidad oculta y/o ofuscada. Alterdroid introduce la un nuevo método de análisis basado en la inyección de fallos y el análisis diferencial del comportamiento asociado.

Por último, presentamos una evaluación del consumo energético asociado a diferentes estrategias de externalización usadas para trasladar a la nube determinadas tareas de seguridad. Para ello, desarrollamos un sistema llamado Meterdroid que permite probar distintas funcionalidades y medir su consumo. Basados en los resultados de este análisis, proponemos un sistema llamado Targetdroid que hace uso de la nube para abordar el problema de la detección de malware dirigido o especializado. Dicho sistema hace uso de modelos estocásticos para modelar el comportamiento del usuario así como el contexto que les rodea. De esta forma, Targetdroid permite, además, detectar de forma automática malware dirigido por medio de estos modelos.

Para finalizar, a partir de las conclusiones extraídas en esta Tesis, identificamos una serie de líneas de investigación abiertas y trabajos futuros basados.

Contents

List of Figures

List of Tables

List of Abbreviations

α	Slope of a given function, page 187
β	Behavioral signature, page 140
$\Delta(\beta_1, \beta_2)$	Differential signature of β_1 and β_2, page 140
γ	Average energy consumption, page 187
$\mathbb{A}$	Set of all relevant and observable activities an app can execute, page 138
$\mathbb{C}$	All possible components of an app, page 136
$\mathbb{O}$	Set of all possible signature transformation operators (STOs), page 139
$\mathbf{g}$	Context of the user during the execution of an app, page 138
$\mathcal{B}$	Set of all possible events for all apps, page 210
$\mathcal{D}$	Dataset composed of feature vectors, page 178
$\mathcal{F}_i$	Set of *apps* belonging to the same family i, page 107
$\mathcal{K}$	Set of clusters, page 123
$\mathcal{M}$	Set of malware families, page 108

r	A cluster, page 123
R_i	A given rule i matching a behavior, page 149
$RD(app)$	Redundancy of a app app, page 107
s	A cluster, page 123
$t_{\text{diffAnalysis}}$	Time of a differential analysis, page 160
t_{exec}	Time during which the app is executed, page 160
$t_{\text{genFaultApp}}$	Time to generate a fault injected app, page 160
u	Inputs provided by a user during the execution of an app, page 138
v	Vector of features, page 115
v_{rs}	A feature vector between derived from r and s, page 123
$w_{i,j}$	Measure of the relevance that the i-th term, m_i, has in object d_j, page 114
y	Intercept with the y-axes of a given function, page 187
z	Vector of features, page 120
A	Transition matrix, page 211
app	Malware sample represented by a sequence of code structure, page 106
M	Markov chain, page 211
N	Length of a given property, page 199
O	Sequence of observed states, page 212

P	An app seen as a collection of components, page 136
t	Time taken to perform a task, page 160
W	Length of a given feature vector, page 198

1

Introduction

Smart devices are rapidly emerging as popular appliances with increasingly powerful computing, networking, and sensing capabilities. Perhaps the most successful examples of such devices so far are smartphones and tablets, which in their current generation are far more powerful than early personal computers (PCs). One of the key differences between such "smart" devices and traditional "non-smart" appliances is that they offer the possibility to easily incorporate third-party applications through online markets (Figure 1.1 depicts such differences).[1]

The popularity of smart devices—intimately related to the rise of cloud-computing paradigms giving complementary storage and computing services—is backed by recent commercial surveys, showing that they will very soon outsell the number of PCs worldwide [Dediu, 2012, 2013]. For example, the number of smartphone users has rapidly increased over the past few years. In 2011, global mobile handset shipments reached 1.6 billion units [Juniper, 2012] and the total smartphone sales reached 472 million units (58% of all mobile devices sales in 2010) [Goasduff and Pettey, 2014].

[1] Although some early feature phones—such as Java ME—allowed the installation of third-party software, their functionality and the support given to both users and third-party developers is relatively limited in comparison to smartphones and other smart devices.

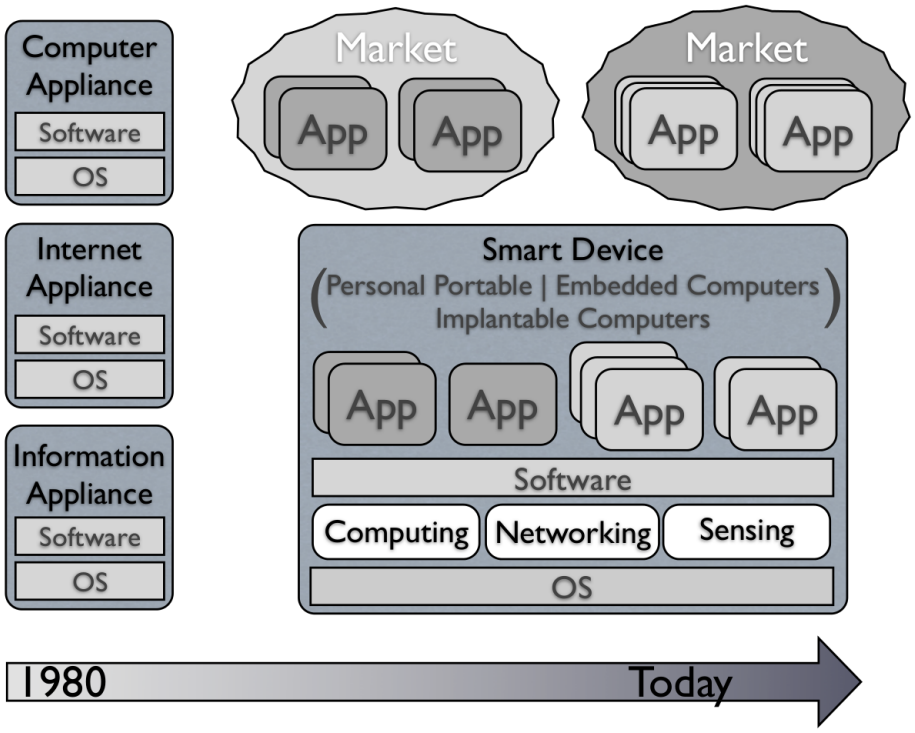


Figure 1.1: Appliances evolution towards smart devices.

In 2012 the smartphone users penetration increased 68.8% and at the end of 2013 reached 1.43 billion units [eMarketer, 2014]. In fact, the number of Android OS and, iOS users is also increasing profusely [Nielsen, 2012]. Specifically, the global mobile OS market share shows that Android OS reached 69.7% at the end of 2012, racing past Symbian OS, BlackBerry OS and iOS as depicted in Figure 1.2. Furthermore, the number of worldwide smartphone sales is expected to keep increasing at least until 2017 [Dediu, 2014b], the average number of applications per device increased from 32 to 41 and the proportion of time spent by users on smartphone applications almost equals the time spent on the Web (73% vs. 81%) [Nielsen, 2012].

New smart devices are appearing at a steady pace, including TVs [Samsung, 2014], watches [Sony, 2014], glasses [Google, 2014b], clothes [CuteCircuit, 2014] and cars [Newcomb, 2014]. This is not only playing a key role in bringing to reality much-

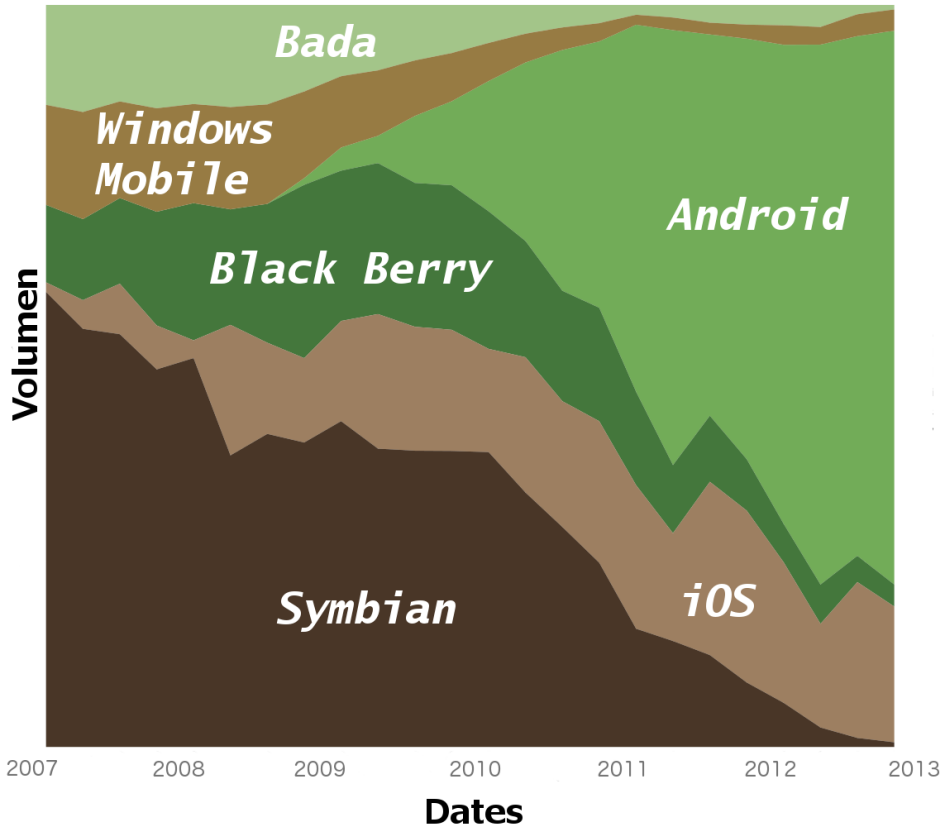


Figure 1.2: Main smartphone platforms by market share from 2007 to 2012 [Dediu et al., 2014].

discussed paradigms such as wearable computing or the Internet of Things (IoT), but also finding innovative and very attractive applications in critical domains such as, for example, healthcare. Both medical staff and patients are increasingly taking advantage of such devices, from regular tablets and smartphones [Larner, 2012] to smart pillboxes [IIH-uBox, 2014], and the new generation of smart wearable systems (SWS) for health monitoring (HM) or implantable medical devices (IMDs) [Chan et al., 2012], among others.

1.1 Smart Malware and Smart Devices

In many respects, smart devices present greater security and privacy issues to users than traditional PCs [Chin et al., 2012]. For instance, most of such devices incorporate numerous sensors that could leak highly sensitive information about users location, gestures, moves and other physical activities, as well as recording audio, pictures and video from their surroundings. Furthermore, users are increasingly embedding authentication credentials into their devices, as well as making use of on-platform micropayment technologies such as Near Field Communication (NFC) [Fenske, 2012].

One major source of security and privacy problems is precisely the ability to incorporate third-party applications, primarily from available online markets but also by other means. There are currently two established models of smart devices according to how users can access such markets [Husted et al., 2011]. In the *open-market* model, users are free to install applications from any online market, whereas the so-called *walled-garden* market model restricts the market from which users can install applications.[2] Many market operators carry out a revision process over submitted apps, which presumably also involves some form of security testing to detect if the app includes malicious code. So far such revisions have proven clearly insufficient for several reasons:

- First, market operators do no give details about how (security) revisions are done. However, the ceaseless presence of malware in official markets reveals that operators cannot afford to perform an exhaustive analysis over each submitted app.

[2] In spite of this, users have found ways of circumventing such restrictions by modifying the device so that other markets will be accessible too.

- Second, determining which applications are malicious and which are not is still a formidable challenge. This is further complicated by a recent rise in the so-called *grayware* [Felt et al., 2011c], namely apps that are not fully malicious but that entail security and/or privacy risks of which the user is not aware.

- Finally, a significant fraction of users rely on alternative markets to get access for free to paid apps in official markets. Such unofficial and/or illegal markets have repeatedly proven to be fertile ground for malware, particularly in the form of popular apps modified (*repackaged*) to include malicious code.

The reality is that the rapid growth of smartphone technologies and its widespread user-acceptance have come hand in hand with a similar increase in the number and sophistication of malicious software targeting popular platforms. Malware developed for early mobile devices such as *Palm* platforms and *feature mobile phones* was identified prior to 2004. The proliferation of mobile devices in the subsequent years translated into an exponential growth in the presence of malware specifically developed for them (mostly Symbian OS), with more than 400 cases between 2004 and 2007 [Dunham, 2008; Shih et al., 2008]. Later on that year, iPhone and Android OS were released and shortly became predominant platforms. This gave rise to an alarming escalation in the number and sophistication of malicious software targeting these platforms, particularly Android OS. For example, according to the mobile threat report published by Juniper Networks in 2012, the number of unique malware variants for Android OS has increased by 3325.5% during 2011 [Juniper, 2012]. A similar report by F-Secure reveals that the number of malicious Android OS apps received during the first quarter of 2012 increased from 139 to 3063 when compared to the first quarter of 2011 [F-Secure, 2012], and by the end of 2012 it already

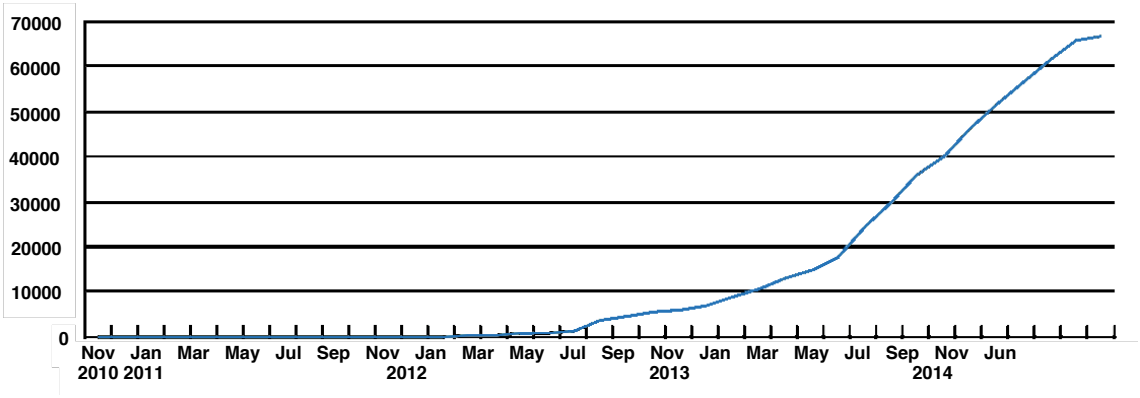


Figure 1.3: Cumulative Android Malware Samples from November 2010 to January 2014 [Sophos, 2014c].

represents 97% of the total mobile malware according to McAfee [McAfee, 2013]. More recently, Sophos Mobile Security Threat Report [Sophos, 2014c] showed that the cumulative Android OS malware samples almost reach 700 thousands reported units by 2014 as depicted in Figure 1.3.

The main factors driving the development of malware have swiftly changed from research, amusement and the search for notoriety to purely economical—and political, to a lesser extent. The current malware industry already generates substantial revenues [Schipka, 2009], and emergent paradigms such as Malware-as-a-Service (MAAS) paint a gloomy forecast for the years to come. This admits a simple explanation from an economic point of view: all in all, attackers seek to minimize the cost required to achieve their goals and, therefore, aim at obtaining the maximum revenues with minimal efforts. For example, the inequality

$$Cost(Attack) < Potential\ Revenue \tag{1.1}$$

is used in [Guido and Arpaia, 2012] to give a cost-benefit analysis of mobile attacks. This fits perfectly the case of smart devices such as smartphones, where malware is

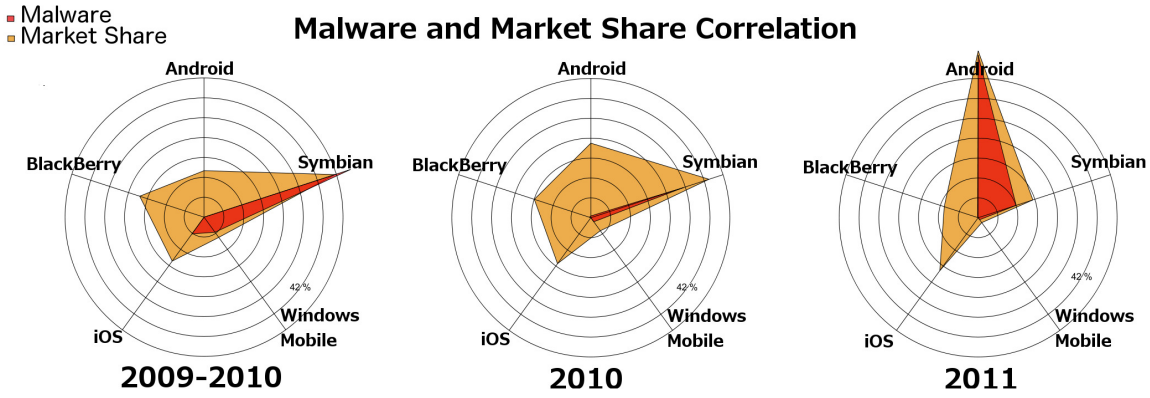


Figure 1.4: Correlation between the number of malware cases and platform market share during a) 2009-2010 [McAfee, 2011], b) 2010 [Juniper, 2012], and c) 2011 [Juniper, 2012].

rather profitable due to (i) the existence of a high number of potential targets and/or high value targets; and (ii) the availability of reuse-oriented development methodologies for malware that make exceedingly easy to produce new specimens. Both points are true for the case of Android OS and explain, together with the open nature of this platform and some technical particularities, why it has become such an attractive target to attackers—see for example Figure 1.4, where the correlation between the market share and the number of unique malware cases reported is straightforward.

Correlations—if not causations—such as those discussed above are paramount to understand future tendencies and threats, not only in the case of smartphones or tablets but also in other devices that soon will likely proliferate. For instance, it has been recently reported that medical devices are plagued with malware [Clark et al., 2013; Vockley, 2012]. Furthermore, it has been shown that RFID-based systems, such as the ones used in several medical devices, are a great infection vector [Rieback et al., 2006]. In the near future, it is quite plausible that similar risks will affect vulnerable IMDs [Burleson et al., 2012], leaving users and patients exposed to exfiltration of highly-sensitive medical information or even malicious manipulation [Halperin et al., 2008a].

Thwarting malware attacks in smart devices is a thriving research area with a substantial amount of still unsolved problems. In the case of smartphones, one primary line of defense is given by the security architecture of the device, one of whose foremost features is a permission system that restricts apps privileges. This has proven patently insufficient so far. For example, in the case of Android OS apps request permissions in a non-negotiable fashion, in such a way that users are left with the choice of either granting the app everything it asks for at installation time or it will not be possible to use it. Most users simply do not pay attention to such requests; or do not fully understand what each permission means; or, even if they do, it is hard to figure out all possible consequences of granting a given set of privileges. For example, applications requesting permission to access the accelerometer of a smartphone or a tablet are rather common. However, it has been demonstrated that it is possible to infer the keys pressed by the user on a touchscreen from just vibrations and motion data [Cai and Chen, 2011]. Thus, using such a permission in conjunction with Internet access—another rather common privilege—could lead to a serious risk of data exfiltration. On top of that, the problem aggravates in platforms where apps can interact with each other and share information, as one needs to consider the privileges acquired by potential collusions.

1.2 Motivation and Objectives

This Dissertation deals with the problem of analyzing smart malware for smart devices, providing specific methods for improving their identification. The Dissertation is strongly biased towards smartphones, since they currently are the most extended class of smart devices and the platform of choice for malware developers and security

researchers. However, our discussion and conclusions apply to other devices as well, and can help to better understand the problem and to improve upon current defense techniques.

We next describe the main motivation and objectives of this work. Firstly, we state that current methods aiming at analyzing smart malware are ineffective and we question the role that security analysts play during the study of large amounts of complex software. Secondly, we establish the need of systematic approaches and automated tools for analyzing smart malware.

1.2.1 Motivation

This Dissertation identifies two fundamental open issues where research is needed: *There is more malware than ever before, and it is increasingly sophisticated*.

P1: Sustained growth in the number of malicious apps targeting smart devices.

As discussed before, malware has become a rather profitable business due to the existence of a large number of potential targets and the availability of reuse-oriented malware development methodologies that make exceedingly easy to produce new samples. The impressive growth both in malware and benign apps is making increasingly unaffordable any human-driven analysis of potentially dangerous apps. This is especially critical as current trends in malware engineering suggest that malicious software will continue to grow both in number and sophistication. As a result, market operators and malware analysts are overwhelmed by the amount of newly discovered samples that must be analyzed. This is further complicated by the fact that determin-

ing which applications are malicious and which are not is still a formidable challenge, particularly for *grayware*.

This has motivated the need for automated analysis techniques and instruments to alleviate the workload of performing intelligent security analysis of software. For instance, when confronted with a continuously growing stream of incoming malware samples, it would be extremely helpful to differentiate between those that are minor variants of a known specimen and those that correspond to novel, previously unseen samples. Grouping samples into families, establishing the relationships among them, and studying the evolution of the various known "species" is also a much sought after application.

P2: Increase in the sophistication of malicious apps and the rise of a new generation of smart malware.

Malware for current smartphone platforms is becoming increasingly sophisticated and developers are progressively using advanced techniques to defeat malware detection tools. On one hand, smartphone malware is becoming more and more stealthy and recent specimens are relying on advanced code obfuscation techniques to evade detection. These techniques create an additional obstacle to malware analysts, who see their task further complicated and have to ultimately rely on carefully controlled dynamic analysis techniques to detect the presence of potentially dangerous pieces of code. On the other hand, the presence of advanced networking and sensing functions in the device is giving rise to a new generation of smarter malware. These malware instances are characterized by a more complex situational awareness, in which decisions are made on the basis of factors such as the location, the user profile, or the presence of other apps.

This state of affairs has consolidated the need for smart analysis techniques to aid malware analysts in their daily functions. This challenge has to be tackled by novel methods to efficiently support market operators and security analysts. In some cases, this problem cannot be solved by market operators alone or by enhanced security models, as they really depend on each user's privacy preferences. For example, a leakage of data such as one's location or the list of contacts might well constitute a serious privacy issue for many users, but others will simply not care about it.

The situation described above inevitably leads to the need for more sophisticated analysis techniques. This, however, poses an important challenge: many devices suffer from strong limitations in terms of power consumption, so certain security tasks executed on the platform may be simply unaffordable. External analysis performed on the cloud in near real time can constitute an alternative. Such a strategy seeks to save battery life by exchanging computation and communication costs, but it still remains unclear whether this is optimal or not in all circumstances. Furthermore, the rise of targeted—user-specific—malware poses one additional challenge: conducting particularized analysis for specific user and execution context.

1.2.2 Objectives

The main goal of this Thesis is to **study methods, tools and techniques to assist security analysts and end users in the analysis of untrusted apps for smart devices and automate the identification of smart malware**.

To achieve this goal, we will focus in the following three general objectives:

- Study the evolution and current state of malware for smart devices, as well as recent progress made to analyze and detect it.

- Develop techniques aiming at better analyzing malware in large scale software markets, with particular emphasis on intelligent instruments to automate parts of the analysis process.

- Facilitate the analysis of complex smart malware in scenarios with a constant and large stream of apps on target. Examples of such sophistication include malware targeting user-specific actions, malware hindering detection with advance obfuscation techniques, or malware exploiting the battery limitations of current devices, to name a few.

1.3 Contributions and Organization

This Thesis provides several contributions in the field of smart malware detection for smart devices aligned with the goals discussed in the objectives above. These contributions are grouped into four related areas, which corresponds to the four central parts of this document: (i) *Foundations & Tools*, (ii) *Static-based Analysis*, (iii) *Dynamic-based Analysis*, and (v) *Cloud-based Analysis*.

Foundations and tools. Part I presents the current state of malware analysis and provides a framework for investigating different analysis and detection strategies for untrusted or malicious code. The following two contributions are presented:

1. **A comprehensive analysis of the evolution of untrusted code for smart devices and current detection strategies.** Chapter 2 provides a characterization of current malware's main features together with an in-depth analysis of both malware and grayware evolution. We identify exhibited behaviors, pursued

goals, infection and distribution strategies, etc. and provide numerous examples through case studies of the most relevant specimens. This chapter also includes a careful review of current detection techniques and presents a taxonomy that provides a comprehensive analysis of their strengths and weaknesses.

The comprehensive study described in in this chapter suggest the need of a versatile and multipurpose research laboratory for smart malware analysis and detection. Thus, Chapter 3 presents a new generation lab and describes the three building-blocks of its architecture: (i) *static-*, *dynamic-*, and *cloud-based analysis* system. Each system is built on a number of open source tools that facilitate the extraction of security features from apps—static features from the apps' components and also dynamic characteristics obtained from their execution. The lab incorporates both physical and virtual devices. These devices are instrumented with cutting-edge tools for monitoring a great number of features: ranging from (i) *hardware-based signals*, such as the battery consumption, to (ii) *kernel-based features* such as the system calls. The lab also includes a dataset composed of a sizable number of apps crawled both from legitimate online markets and malicious public and private repositories. This new generation lab is shown to be paramount for the evaluation of all contributions presented in this Thesis, and extremely useful for automating malware analysis for smart detection.

Static-based Analysis: Part II exploits the use of static features to assist the security analyst in the large scale analysis of malware families:

2. **A text mining approach for analyzing and classifying malware families.** Chapter 4 analyzes several statistical and semantic features to facilitate the

identification of malicious code components and their similarity to other apps. This Chapter shows how static analysis can be used to classify malware with a technique named Dendroid. Dendroid a system based on text mining and information retrieval techniques used for automating parts of the malware analysis process. This approach is motivated by a statistical analysis of the code structures found in a dataset of Android OS malware families, which reveals some parallels with classical problems in information retrieval domains. To this end, we adapt the standard Vector Space Model [Salton et al., 1975] and reformulate the modeling process followed in text mining applications. This enables us to measure similarity between malware samples, which is then used to automatically classify them into families. We also investigate the application of hierarchical clustering over the feature vectors obtained for each malware family. The resulting dendrograms resemble the so-called phylogenetic trees for biological species, allowing us to conjecture about evolutionary relationships among families. In fact, this contribution reveals that current malware families abuse from a reuse-oriented development methodology, which boosts static-based detection strategies.

Dynamic-based Analysis. Part III compiles efforts based on the dynamic execution of untrusted code and the analysis of its resulting behavior. The following fundamental contribution is tackled:

3. **Differential fault analysis of obfuscated malware behavior.** Obfuscated malware provides attackers with the ability to evade static analysis. Chapter 5 introduces a dynamic-based detection technique called Alterdroid for identifying obfuscated malware on large-scale analysis scenarios. Alterdroid provides

security analysts with a framework capable of automating the identification of obfuscated components distributed as parts of an app. The key idea in Alterdroid consists of analyzing the behavioral differences between the original app and a number of automatically generated versions of it where a number of modifications (faults) have been carefully injected. Observable differences in terms of activities that appear or vanish in the modified app are recorded, and this signature is finally analyzed through a pattern-matching process driven by rules that relate different types of hidden functionalities with patterns found in the differential signature.

Cloud-based Analysis. Part IV contains two contributions related to the use of the cloud to offload detection strategies from devices. The first contribution explores the question of offloading—or not—general anomaly-based detection strategies. The second contribution stands over the conclusions extracted from the first one, and approaches the detection of targeted malware using a cloud-based strategy. We next summarize each one:

4. **Power-aware anomaly detection in smartphones.** Many recent works simply assume that on-platform detection is prohibitive and suggest using offloaded (i.e., cloud-based) engines. Chapter 6 studies different security tasks involved in the detection of malware in built-in detection systems. Specifically, it focuses on machine learning based anomaly detection systems, as they are widely used to build both static and dynamic detection techniques. This chapter studies the power-consumption trade-offs among different strategies for off-loading, or not, those security tasks. It also shows that outsourced detection strategies are clearly the best option in terms of power consumption when compared to

on-platform detection. This contribution also points out noticeable differences among different machine learning algorithms, and provides separate consumption models for functional blocks (data preprocessing, training, test, and communications) that can be used to obtain power consumption estimates and compare detectors.

5. **A stochastic behavioral-triggering model for targeted malware detection.** Targeted malware challenges current dynamic-based detection strategies as analysts must reproduce very specific activation conditions to trigger malicious payloads. Furthermore, the consumption model presented in Chapter 6 shows that the use of detection techniques built in the device is unaffordable. Chapter 7 proposes a cloud-based system, called Targetdroid, to facilitate the detection of this type of malware. The contribution presented in this chapter relies on automatically learned stochastic models of usage and context events derived from real users. This chapter reveals several interesting particularities of apps usage patterns that allow for an efficient generation of testing patterns. This contribution shows that testing patterns automatically is feasible, specially when this is done in conjunction with a cloud infrastructure.

Finally, Part V presents the main conclusions, analyzes the contributions of this Thesis and the published results, and discusses open research problems and future work. This part also comprises the references and appendices.

Foundations and Tools

2

Evolution, Detection and Analysis of Malware for Smart Devices

2.1 Introduction

This chapter presents a comprehensive study of the evolution and current state of malware for smart devices and techniques proposed to thwart malware attacks. We first describe current smartphone security architectures and discuss a number of research works that have recently proposed enhanced models to provide protection against malicious applications (see Section 2.2). We then provide in Section 2.3 a characterization of the various categories of malware developed for smart devices by identifying possible attack goals, distribution and infection strategies, and exhibited behavior. Other authors (e.g.,[Felt et al., 2011c; Zhou and Jiang, 2012]) have previously discussed similar issues for smartphone malware, but not to the extent covered by this work. Furthermore, our taxonomy is used to analyze the evolution of malware using a representative sample of specimens that have gained notoriety over the last few years. Finally, Section 2.4 analyzes and discusses malware detection

approaches specifically developed for smart devices. Again, we first identify a number of features according to which each technique can be classified and use them to provide a systematic review of the most relevant works proposed so far. Among our contributions, we identify an extensive number of indicators that can be monitored to detect the presence of malware and that apply to any kind of smart device—not only smartphones or tablets. Additionally, we correlate these features with our malware characterization, pointing out how each class of malicious behavior manifests in terms of observable indicators.

2.2 Security Models in Current Smart Devices

In this section we provide an overview of the security models and protection measures incorporated in current smart devices, with particular emphasis on smartphones. The two major mobile platforms—iOS and Android OS—are built upon traditional desktop Operating Systems (OS) and inherit some security features from them. However, they also employ more elaborated security models designed to better fit the architecture and usage of these devices.

2.2.1 Security Features

A number of recent works (e.g., [Asokan et al., 2013; Enck, 2011; Kostiainen et al., 2011; Li and Clark, 2013]) have provided detailed account of the major security features incorporated in smartphones. In what follows we restrict ourselves to highlight the fundamentals about:

1. security measures implemented at the market level;

2. security features incorporated in the platform; and

3. an overview of recently proposed security mechanisms

with particular emphasis on the protection against malware that they provide.

2.2.1.1 Market Protection

A primary line of defense against malicious software consists of preventing it from entering available distribution markets. To this end, two basic security measures are applied at the market level:

- **Application review.** Some official markets analyze submitted apps before making them available for download and install. Operators do not give details about the particularities of such reviews, but it is generally understood that some form of security testing is carried out. Furthermore, in walled-garden models devices can only access some markets, which presumably only distribute reviewed apps.

- **Application signing.** Most markets force authors to sign their apps. This allows authors to claim authorship and also has some technical consequences in certain platforms (e.g., apps signed with the same certificate can share resources). Thus, a device can be sure about the integrity of an app by verifying the associated signature against the corresponding certificate authority.

Both measures have proven so far insufficient to combat malware. Manually reviewing applications is a a difficult and time-consuming task, impossible to perform in full extent due to the massive number of applications being submitted every

day. Automated approaches have been recently explored as an affordable alternative [Batyuk et al., 2011; Gilbert et al., 2011; Lockheimer, 2014; Zhou et al., 2012b]. For instance, in 2012 Google announced an application approval tool named Google Bouncer [Lockheimer, 2014] for Android OS. Also in this line, Zhou et al. proposes *DroidRanger* for detecting smartphone malware in Android markets [Zhou et al., 2012a,b]. Their analysis shows that the infection rate in alternative marketplaces is one order of magnitude higher than the official marketplace. Additionally, they found that about 0.1% of the 204.040 analyzed applications are malicious. We however believe that such a fraction is much higher for two reasons. On the one hand, samples were taken during a two-month period in the first and third quarter of 2011. However, according to McAfee Threat Report [McAfee, 2012], the number of Android OS malicious samples experimented an exponential growth of 400% during the fourth quarter of that year. On the other hand, the detection heuristics used by authors present a high false negative rate, ranging from 5.04% to 23.52%.

Even if application review processes were perfect, many devices install applications through unofficial markets in which there are no guarantees whatsoever about the trustworthiness of such apps. Application signing can give users some assurance about the integrity of software downloaded from a questionable source, particularly when such software claims to be an unmodified copy of the same available in official markets. But most of the time users do not perform such verifications, nor it is possible to do so in many cases as signatures are stripped off.

2.2.1.2 Platform Protection

Current platforms incorporate a number of mechanisms to confine and limit the actuation of malicious apps once installed in the device:

- **Permissions**. Most platforms provide a permission-based system aimed at restricting the actions that an app can execute on the device, including access to stored data and available services (e.g., networking, sensors, etc.). Au et al. [Au et al., 2011] examine the permission system of several smartphone OS, focusing on:

  1. The amount of control users have over app permissions. Depending on the granularity offered by the OS, users can grant privileges using precise or coarse permissions. Additionally, such permissions cannot always be individually enabled or disabled.
  2. The information they convey to the user. Several platforms offer the users specific information about how applications are using resources. While some OS only inform of what resources the application may use, others track the actual use of permissions throughout execution.
  3. The interactivity of the system. Some permission systems require a heavy intervention of the user. Typically, fine-grained permissions require more interaction than coarse-grained. Furthermore, permissions can either be requested only once (assuming they will remain the same) or they can be requested periodically.

  A summary of their analysis is shown in Table 2.1. These results will be further discussed later on Section 2.2.2 when discussing the security features of the

Platform	#Perm.	Control	Information	Interactivity
Android OS	75	Medium	High	Low
Windows Mobile	15	Medium	Medium	Low
iOS	1	Low	Low	Low
BlackBerry OS	24	High	High	High

Table 2.1: Permission models in the main Smartphone platforms [Au et al., 2011].

most important platforms. A recent study by Felt et al. [Felt et al., 2011b,d, 2012] on the effectiveness of app permission systems concludes that they are rather effective at protecting users. However, in the case of Android OS it points out that many apps request a significant amount of permissions identified as potentially dangerous and that frequent exposure to warnings drastically reduces effectiveness. Furthermore, authors also conclude in [Felt et al., 2011b] that apps are often over-privileged due to a lack of documentation and development bad practices. In this regard, Barrera et al. [Barrera et al., 2010] propose a methodology for analyzing permission-based security models and suggest to increase the expressiveness without maintaining the total number of permissions.

- **Sandboxing**. Sandboxing is a security mechanism used by some platform architectures to isolate running applications based on mandatory access control policies. Sandboxing can provide protection against malicious applications to a certain extent, but are ineffective if users overlook the permissions entitled to installed apps [Felt et al., 2012]. Furthermore, sandboxing do not prevent apps from exploiting system or kernel vulnerabilities and, besides, can also be bypassed in some cases [Davi et al., 2011a]. In this regard, several works [Andrus et al., 2011; Gudeth et al., 2011; Husted et al., 2011; Lange et al., 2011; Wu et al., 2014] propose the use of hypervisors that run directly on the hard-

ware. Other authors (e.g., [Russello et al., 2012]) have focused on optimizing the virtual machine manager, as virtualization introduces a trade-off between security and performance [Xu et al., 2010].

- **Interactions between apps**. Some platforms provide the developer with a rich inter-application communication system to facilitate component reuse. Such Inter Component Communication (ICC) systems introduce several security issues. For example, in a compromised device messages exchanged between two components could be intercepted, stopped, and/or replaced by others, as they generally are not encrypted or authenticated. Additionally, two or more malicious applications can collude to violate app security policies, such as for example in the so-called re-delegation attacks [Felt et al., 2011a]. Chin et al. [Chin et al., 2011] have recently identified a number of security risks derived from the app interaction system in Android OS. Their reported results show that 97% of the analyzed applications are exposed to activity hijacking; 57% to activity launch; 56% to broadcast injection; 44% to broadcast theft; 19% to service hijacking; 14% to service launch; and 13% to system broadcast without action check.

- **Remote management**. Some market and network operators, as well as platform manufacturers, are empowered with the ability to remotely remove apps from the device and even repair damages caused by malware. This can be seen as an extension of other functionalities already present, such as for example updating the OS or applying patches. However convenient, this feature can be seen by many users as too intrusive and is not exempt from risks, both privacy-wise but also in case of compromise of the remote management function.

2.2.1.3 Other Proposals

Over the last few years there has been an explosion of proposals suggesting enhanced security models and alternative policy languages to improve upon the limitations discussed above. The interested reader can find a summary in recent surveys, such as for example [Enck, 2011]. The majority of them fall in one or more of the next categories:

1. **Rule driven policy** approaches [Bugiel et al., 2011a; Conti et al., 2011; Enck et al., 2009a; Ongtang et al., 2009; Titze et al., 2013] propose richer languages based on rules, aiming at palliating insufficient policy expressibility on current protection systems.

2. **High-level policy** protection techniques focus on enforcing information flow throughout the system. Several approaches focus on applying different labeling systems [Mulliner et al., 2006], while others enforce full isolation based on distinct security profiles [Russello et al., 2012] or policies [Russello et al., 2013] within a single device.

3. **Platform hardening** aims at simplifying underlying platform layers, i.e., boot-loader and kernel, to mitigate the risk of unpatched vulnerabilities [Husted et al., 2011]. SELinux-based systems [Shabtai et al., 2010a] and remote attestation [Nauman et al., 2010] approaches can be applied to improve trusted computing base protection.

4. **Multiple-users** protection assumes scenarios where different users share the same device. Several approaches focus on applying different access control

mechanisms such as DifUser [Ni et al., 2009] or RBACA [Rohrer et al., 2012] (a Role Based Access Control for Android).

Most of these proposals would certainly provide enhanced protection against malicious apps. However, in many cases they ultimately rely on richer—and more complex—policies that users must specify. But users generally lack security expertise [Kraemer and Carayon, 2007], and developing complete and consistent security policies is far from being an easy task even for experts with the appropriate background. It can be argued that devices could use policies created by others, but it is unclear to what extent "one size fits all." Furthermore, there is an incipient interest on intentionally bypassing the platform protection mechanisms to gain full control of the device and, for example, install apps otherwise forbidden.

2.2.2 Security Features in Dominant Platforms

When compared with traditional PCs, smartphone platforms have taken an innovative approach to securing the device and the distribution of software. We next provide an overview of some of the security features present in the five platforms that currently dominate the market.

2.2.2.1 Symbian

Symbian OS security model is based on a basic permission system. Phone resources are controlled by the OS using a set of permissions called "capabilites". Furthermore, applications run in user space, while the OS run in kernel space. Those applications requiring access to protected resources must be signed by Symbian or the device

manufacturer, while all others can be self-signed [Kostiainen et al., 2011]. There is very little information about protection at the market level.

2.2.2.2 BlackBerry

BlackBerry security model is based on a coarse-grained permission protection model. Applications have very limited access to the device resources and, as in the case of BlackBerry OS, they must be signed by the manufacturer (RIM) to be able to access resources such as, for example, the user's personal information. Additionally, applications must get user authorization to access resources such as the network. However, once the user grants access to an application to use the network, the application can both send SMSs and connect to Internet [O'Connor, 2006]. Although applications are not executed in a sandbox, some basic process and memory protection is offered. For instance, a process cannot kill other processes nor access memory outside the app bounds.

2.2.2.3 Android

Google's Android OS security model relies on platform protection mechanism rather than on market protection, as users are free to download applications from any market. Applications declare the permissions they request at installation time through the so-called manifest. If the user accepts them, the operating system will be in charge of enforcing them at running time.

Many researchers have pointed out that Android OS's permissions are overly broad and have proposed alternatives and extensions [Fang et al., 2014]. For example, Ongtang et al. propose a fine-grained permission model called *Saint* to limit the

granularity at which resources are accessed [Ongtang et al., 2009]. Similarly, Jeon et al. [Jeon et al., 2011] propose a framework that enhances Android OS's security policies and extends permission enforcement both an installation time and during runtime. Schreckling et al. introduced in [Schreckling et al., 2012] *Constroid*, a framework to define data-centric security policies for access management. Security policies are here defined for each individual resource, instead of specifying permissions for each app. Furthermore, such definition can be done at a fine-grained level, allowing users to, for example, grant an app access to a part of the address book only. A major consequence is that security policies are therefore defined by the user, not by the developer. However, this approach can easily overwhelm users as they are held responsible of specifying security and privacy policies.

Additionally, Android OS uses sandboxing technique and Address Space Layout Randomization (ASLR) to protect applications from malicious interference of others apps. Although Android OS isolates each running process, apps can still communicate with each other using ICC, a rich functionality that, however, introduces risks such as those discussed before. Bugiel et al. introduce a security framework called TrustDroid [Bugiel et al., 2011b] to separate trusted an untrusted applications into domains, firewalling ICCs among these domains. Similarly, Dietz et al. propose *Quire* [Dietz et al., 2011], a signature scheme that allow developers to specify local (ICC) and remote (RPC) communication restrictions. Other proposals such as TaintDroid [Enck et al., 2010], AppFence [Hornyack et al., 2011] or XManDroid [Bugiel et al., 2011a] closely monitors apps to enforce given security policies. The first two uses dynamic taint analysis to prevent data leakage and protect user's privacy, while the last one extends Android OS's security architecture to prevent privilege escalation attacks at runtime. The main difference between TaintDroid and AppFence is that

the latter tries to covertly anonymize private information prior to blocking leakages.

Furthermore, all Android OS applications must be signed with a certificate to identify the developer. However, the certificate can be self-signed, in which case no certificate authority verifies the identity of the developer.

Several articles discuss Android OS security model [Enck et al., 2009b; Shabtai et al., 2010b], providing a deep understanding of android architecture. Enck et al. [Enck et al., 2011] also present a study of Android security by analyzing 1100 free applications. We refer the reader to these works for further details.

2.2.2.4 iOS

Apple's iOS security model [Apple, 2012] relies on market protection mechanisms rather than enforcing complex permission polices on the device at installation time. Apple's App Store is a walled-garden market with a rigorous review process. Those processes are essential for preventing malware from entering the device, as runtime security mechanisms are limited to sandboxing and user supervision. iOS isolates each third-party application in a sandbox. However, most of the device's resources are accessible[1] and misuse of a few of them—such as GPS, SMS, and phone calls—can only be detected by the user after installation.

Specific details on Apple's App Store application review are unknown. In July 2009 Apple revealed that at least two different reviewers study each application [Apple, 2014]. However, it is probable that Apple uses also static and dynamic analyses.

[1] In iOS version 5, although Apple is likely to introduce some modifications in iOS version 6. Specifically, the new version will restrict access to most of the device's resources [Chubb, 2014].

Applications distributed on Apple's App Store must be signed by a valid certificate issued by Apple. Developer certificates are issued to individuals and/or companies after obtaining a verified Apple credential. iOS dynamically verifies that the application is signed, and therefore it is trusted, before executing it. Nevertheless, iOS can be tampered with (jailbroken) to install applications from alternative markets. This practice violates Apple policies, causes the device to lose its warranty, and allows the distribution of piggyback malware repackaged together with the original app.

Latest versions of iOS provide a number of features to protect user data based on master encryption keys and protected by a passcode. The entire file system is encrypted using block-based encryption and can only be decrypted when the phone is unlocked. Additionally, iOS supports ASLR and Data Execution Prevention (DEP) to prevent the execution of arbitrary code at runtime.

2.2.2.5 Windows Mobile

Microsoft's market protection model for Windows Mobile systems is based on application review. Developers are also validated prior to application's approval. Platform protection in Windows Mobile is similar to Android OS. It uses a trusted boot component and code signing to protect the integrity of the operating system. It also provides signed drivers and applications through the *Windows Phone Store* online market.

Latest versions of Microsoft's smartphone OS (Windows Phone 7 and 8) incorporate isolation among different sandboxes [Microsoft, 2012], and each app is executed in its own sandbox, named "chamber". Chambers are defined and implemented using system policies, which restrict the access to other chambers. While chambers

are defined and implemented using a number of system policies, each security policy defines what permissions are given to an app, known as capabilities. In this regard, users are informed of the capabilities of an application prior to install.

2.3 Malware in Smart Devices: Evolution, Characterization and Examples

Malicious applications for smart devices—notably smartphones—have rocketed over the last few years, evolving from relatively simple apps causing annoyance to complex and sophisticated pieces of code designed for profit, sabotage or espionage. In this Section we first provide a brief overview of such evolution from early mobile platforms to current devices. We subsequently propose a number of features that can be used to classify, characterize and better understand malware for smart devices.

2.3.1 Evolution

As in the case of traditional PCs, where malware evolution was intimately connected to the increase in computing resources and the advent of the Internet, the complexity and hostility of malicious software has intensified from early mobile handsets to the current generation of smart devices. In the early 2000s, *Palm* platforms were affected by malicious software that mimicked strategies well-known in PC malware. For example, *Symb/Liberty*, *Symb/Vapor* and *Symb/Skuller* were popular Trojans at the time, i.e., applications that perform some useful function while simultaneously conducting malicious activities. Others such as *Symb/Phage* employed classical virus propagation strategies to infect additional programs present in the handset. Their

malicious payload varied, but in all cases it was sought to inflict damage over user information or corrupt system files in order to cause a device failure.

The rise of *feature mobile phones* brought about a variety of distinctive infection vectors when compared to traditional PCs, primarily through the communication and networking functions offered by 3G, Wi-Fi, EDGE, Bluetooth, the SMS/MMS messaging system, and NFC [Fleizach et al., 2007; Verdult and Kooman, 2011]. For instance, *Symb/Cabir* was one of the first Symbian OS worms using Bluetooth to infect other devices. Additionally, when handsets were given Internet connectivity and the possibility to easily install third-party applications, more sophisticated infection strategies appeared. One early example was *Symb/Yxes*, which used the SMS channel and support from remote servers to propagate and configure itself.

The availability of mobile networking and pay-per-use services contributed to a rapid escalation of the malware phenomenon, both in feature phones and smartphones. Examples such as *Android/YZHCSMS.A* and *WinCE/Fakemini* send premium-rate SMSs without the user's knowledge, which results in very significant revenues for the owner of the registered number. Others such as *Android/Smspacem* have been also driven by economic incentives: sending spam through SMSs.

In recent years, the proliferation of smartphones with improved sensing and networking capabilities has translated into more sophisticated threats. For example, *Android/DroidKungFu* and *iPhone/FindAndCall* steal a variety of personal information stored in the device and exfiltrate it through the network to a remote server. Other pieces of malware such as *Android/Spybubble*, *Android/Nickispy* and *FinSpy Mobile*[2] have evolved into fully fledged spy instruments with the ability to monitor,

[2]FinSpy is a surveillance component part of a commercial surveillance toolkit called *FinFisher*, designed to spy over a wide range of mobile platforms. The mobile version is capable to monitor apps, emails, text messages, etc. on Android, iOS, BackBerry, Symbian, etc.

record and exfiltrate the device's current location, ongoing and past phone calls and SMS logs to name a few. Although more illustrative examples are provided later on this section, readers interested in a more in-depth study are referred to the recent work of Zhou and Jiang [Jiang and Zhou, 2013; Zhou and Jiang, 2012], where a study of more than 1200 malware samples is presented.

It is plausible to believe that similar threats will soon affect other smart devices such as smart TVs or IMDs. For example, Auriemma [Auriemma, 2014] has recently shown that several versions of Samsung's Smart TV [Samsung, 2014] are vulnerable to buffer-overflow attacks that could allow an attacker to remotely control the device. Many security vendors are already releasing security frameworks for smart TVs, including antimalware products [Sophos, 2014a]. The situation may become similar for medical devices too, particularly for those designed to remotely monitor a patient's condition and/or control body functions. We are only aware of a few cases of malware reported so far that affects existing IMDs or other medical smart devices [Clark et al., 2013], although researchers believe that malicious programs will certainly rock soon [Clark et al., 2013; Halperin et al., 2008b; Vockley, 2012].

2.3.2 Malware Characterization

Current malware for PCs have evolved into complex and reuse-oriented pieces of software. Traditional classifications have focused on factors such as the propagation strategy (e.g., viruses vs. worms) or the malicious activity carried out (trojan horses, spyware, adware, rootkits, etc.), among others [F-Secure, 2014; Felt et al., 2011c; Symantec, 2014; Zhou and Jiang, 2012]. However, these categories are rather imprecise and do not contribute to a better understanding in terms of detecting the

presence of malware, particularly in current times where most malware present multiple and constantly changing features.

We next identify several criteria according to which malware in smart devices can be described and classified. Each provided criterion will be subsequently associated with some observable behavior in one or more features of the device. Thus, our classification will serve both to better understand the functionality of malware, but also to point out where to look for detecting malicious activities. We believe this can be of help to improve upon current detection strategies.

We classify malware for smart devices in terms of the following three features (a graphical summary is provided in Figure 2.1):

- **Attack goals and behavior**: Identifying malware's motivation on smart devices is paramount to have a better understanding of its behavior and can be used to develop targeted detection strategies. Such goals range from fraud and service misuse driven by economic incentives, to spamming, espionage, data theft and sabotage.

- **Distribution and Infection**: Malware creators can use a variety of techniques to distribute malicious applications and infect devices, from self-propagation mechanisms based on vulnerabilities and misconfigurations, to simply tricking the user into installing it by means of social-engineering techniques.

- **Privilege acquisition**: Once the malicious code is installed on the device, it often needs to acquire enough privileges to carry out its goals. This is automatic in many cases, as the user might already have granted them to the app, whereas in other cases technical vulnerabilities and/or misconfigurations are exploited.

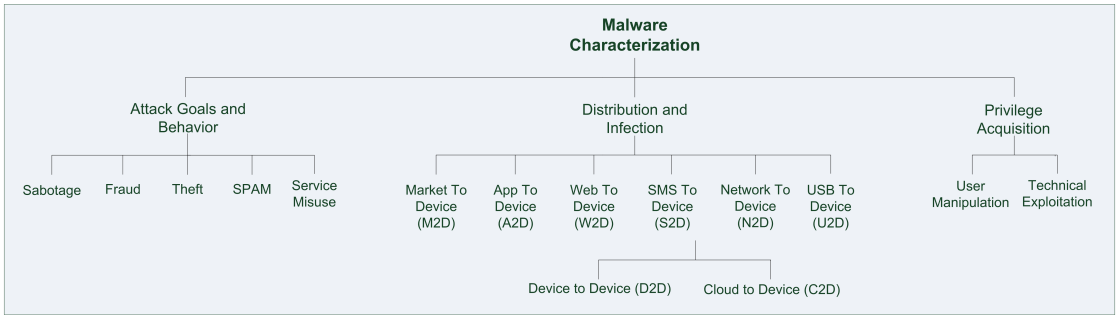


Figure 2.1: Malware characterization for smart devices.

In the remaining of this section we describe each criterion in detail and discuss some illustrative examples.

2.3.3 Attack Goals and Behavior

Felt et al. [Felt et al., 2011c] analyze the main incentives behind iOS, Android OS, and Symbian OS malware using a dataset containing 46 specimens found between 2009 and 2011. According to their analysis, the most common malicious activities are related to the exfiltration of personal information and user credentials (44%), followed by premium-rate SMSs (33%) and, to a lesser extent, research, novelty, or amusement purposes. It is also pointed out that the majority of the analyzed pieces exhibited behaviors related to more than one incentive, and that they often incorporate secondary goals such as SMS advertisement, spamming, search engine optimization and, in a few cases, ransom. About the 33% of the studied malware changed their behavior based on commands received from a Command and Control (C&C) server.

More recently, new pieces of malware such as *Android/NotCompatible* [Lookout, 2014] are demonstrating that attackers' interests are not only limited to the scope of a smartphone and its user, but to large private networks. By turning an

infected device into a TCP relay/proxy—capable of forwarding network traffic—, smartphones can be used to support many infection vectors. For instance, an attacker could establish an encrypted point-to-point session via HTTP with a device located behind the firewall. Using such tunnel, the attacker might be able to probe the private network and run exploits against assets within the corporation. Thus, malware such as *Android/NotCompatible* opens new opportunities for penetrating corporate networks.

Understanding the motivations behind malware can lead to a better identification of its behavior. Figure 2.2 presents the relation between most common incentives and the behavior associated with them. Common behaviors can be classified in *monitoring* (eavesdropping, profiling, etc.), *service misuse* (SMS, call, email, other services used for spamming, etc.), *sabotage* (draining the battery, deleting critical files, etc.), *data exfiltration*, and *fraud*. Note that some behaviors could affect two or more categories. For example, the unauthorized use of SMSs for spamming might well be both a service misuse and a fraud.

2.3.3.1 Example: Smartphone-based Botnets

A botnet is a collection of compromised devices that can be remotely controlled by an attacker (i.e., the bot master). As the number of smartphones is rapidly approaching the number of PCs, botnets for such platforms have gained momentum using a variety of distribution strategies to harvest as many devices as possible.

Traynor et al. [Traynor et al., 2009] were among the first to study the potential theoretical impact of mobile-phone botnets in cellular networks. As far as we are aware, the first mobile botnet—named *SymbOS/Yxes*—appeared in 2009 and

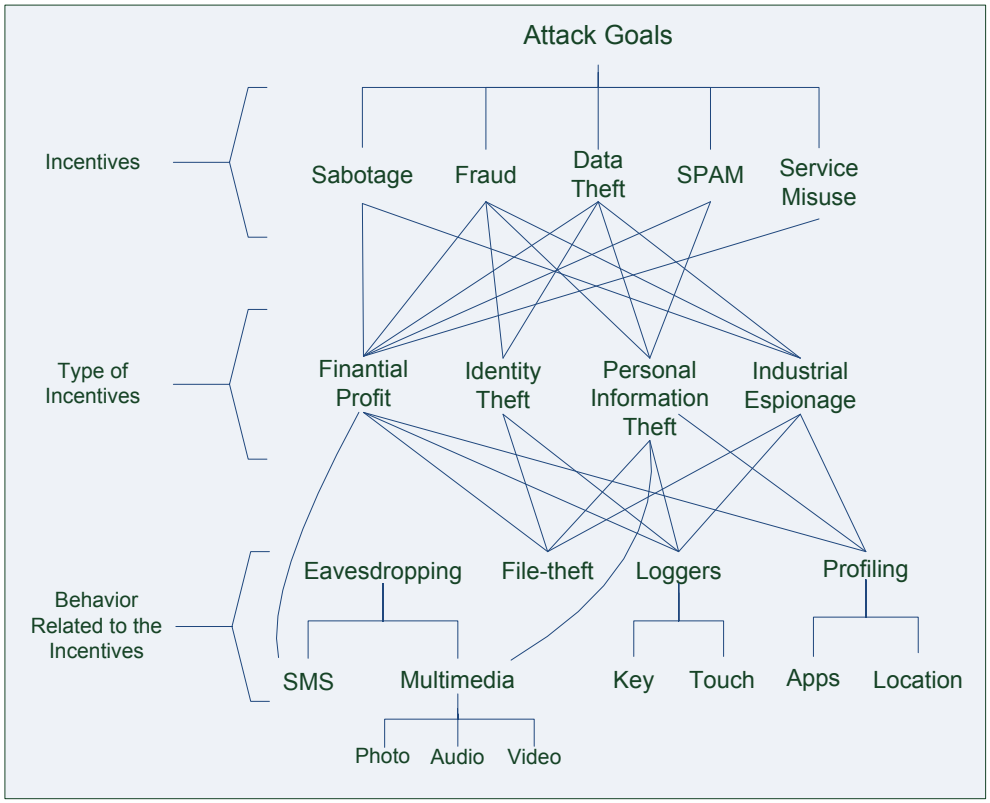


Figure 2.2: Main attack goals, associated incentives, and exhibited behavior for malware in smart devices.

targeted Symbian OS platforms, using a rudimentary HTTP-based command and control (C&C) channel. *iPhone/Ikee* appeared later on that same year, infecting around 21000 iPhones within two weeks. One remarkable feature of *Ikee* was that it showed how easy it can be to hijack a smartphone platform when root exploits are available. Specifically, it exploited iPhones that were left with the SSH port open and a default password after having been jailbroken. Such simple but very effective attack vectors can enable an attacker to control thousands of devices through an easy-to-implement C&C mechanism, as *Ikee.B* did [Porras et al., 2010].

C&C resilience is essential for a botnet to survive. In this regard, smartphones are very attractive devices, as they offer multiple communication alternatives that can be

leveraged to implement a C&C channel, including rather non-standard means such as SMSs [Mulliner and Seifert, 2010]. Mulliner et al. implemented and evaluated an iPhone-based mobile botnet named *iBot* and demonstrated that thwarting them is more challenging than in computer networks, in particular because of employing multiple C&C channels (HTTP, SMS, etc.) in a peer-to-peer (P2P) fashion.

Android/Andbot [Xiang et al., 2011] introduced a new energy-aware C&C strategy named URL Flux for Android OS botnets. *Android/Andbot* uses URL Flux to eliminate the single point of failure problem present in *Ikee.B* and also reduces the SMS fees incurred by *iBot*. URL Flux is a domain name conversion used by *Confiker*—a Windows worm that infected millions of computers between 2009 and 2011—based on a domain generation algorithm seeded with a public key. Recently, more advanced iOS rootkit-like malware such as iSAM [Damopoulos et al., 2011] integrates multi-functional tools also capable of self-propagating to other iPhone devices in ways similar to *Ikee*'s.

Obfuscation is becoming popular in botnets, both by encrypting communications exchanged over the C&C channel and also local resources that might facilitate detection through static analysis, such as server names and URLs, keywords, file names, etc. *AnserverBot* makes extensive use of some of these techniques, and also relies on posts made on public blogs to retrieve code updates and communicate with other members of the botnet.

2.3.3.2 Example: Grayware

The so-called grayware apps gather potentially sensitive user and/or device information, sometimes without user knowledge, and use it for dubious purposes or in

contexts that the user might well not approve. For example, *Aurora Feint* is an app that sends the whole address book to an unknown destination and was quickly delisted from Apple's market in July 2008. Similarly, the author of *Storm8*—a popular game—was sued for collecting users' phone numbers, and *Twitter* has been widely criticized for sending the phone's contact list without informing the user.

Most grayware apps claim to retrieve such information for legitimate purposes and that it is crucial to improve the quality of the service offered to users. This, however, has recently become a major privacy threat for users' privacy, as apps collect excessive amounts of personal information and it remains unclear whether the service provider will use that data for legitimate purposes or not. Some platform manufacturers are increasingly deploying measures to prevent this. For example, in iPhone a strict control is carried out to guarantee that personal information is not sent to the cloud unless really needed.

2.3.4 Distribution and Infection Strategies

Malicious programs employ a number of distinctive techniques to distribute themselves. We next discuss the most relevant and propose a taxonomy to classify them according to the channel used to enter the device. Distribution techniques are primarily influenced by malware in desktop computers, although the emergence of app markets have opened new possibilities. Two main approaches exist: (i) self-propagation and (ii) social engineering. A self-propagating piece of malware can use different strategies to automatically install the payload into a device, whereas social engineering-based distribution strategies exploit the security unawareness of users to trick them into manually installing the application (e.g., *Andr/Opfake-C* by Sophos

[Sophos, 2014b], which spreads via Facebook and, once installed, allows the attacker to perform premium-rate calls).

We have identified six different distribution vectors that can be used to infect devices:

- **Market to Device (M2D)**: This propagation strategy is based on market-borne attacks. An attacker uploads a malicious application to a market, sometimes using a stolen identity. Users can only get infected if markets accept such malicious apps and users install them. Open markets, in particular those performing little or no security revisions, are particularly vulnerable to this distribution method. For instance, malware using devious exploits (e.g.: *Android/DroidKungFu*[3]), might compromise the device by these means.

- **Application to Device (A2D)**: This propagation strategy is based on application-borne attacks. An attacker might rely on a specific, vulnerable application to spread itself. For instance, instances such as *Andr/Opfake-C* can use Facebook to post links with a copy of the malicious code. The main difference with M2D is that attackers assume the presence of other installed applications (presumably "goodware") to achieve infection. In this regard, even walled-garden models can be vulnerable to this type of infection vector.

- **Web-browser to Device (W2D)**: W2D uses web-borne attacks to propagate the malware in way similar to A2D. In this regard, we can consider W2D an specific type of A2D. The difference is that A2D strategies are limited by the possibilities offered by the application, whereas in W2D malware can exploit

[3] *Android/DroidKungFu* uses an exploit called 'Rage Against The Cage" [Kramer, 2010] for privilege escalation

general drive-by-download strategies. This attack vector has recently gained popularity due the widespread use of vulnerable multi-platform components such as WebView [Luo et al., 2011].

- **SMS to Device (S2D)**: This strategy is used by malware that propagates via SMS or MMS or attacks that distribute a malicious payload by these means.
- **Network to Device (N2D)**: This propagation strategy is based on exploiting vulnerabilities or misconfigurations in the device. We distinguish between:
  - **Device to Device (D2D)**: When distribution is driven by another device in a P2P-fashion, and
  - **Cloud to Device (C2D)**: When distribution is done by a powerful computer such as a workstation or a server.
- **USB to Device (U2D)**: This strategy is used by malware that enters the device through a port (typically a cable) when connected to an infected PC.

2.3.4.1 Example: Repackaging

One of the most common distribution strategy for smartphone malware consists of repackaging popular applications and distributing them through alternative markets (M2D) with additional malicious code attached. Repackaging is not a phenomenon exclusive of the current generation of smartphones, although the proliferation of these platforms and the impressive growth in available apps have certainly contributed to make it a popular infection strategy. As far as we know, M2D repackaging started with Symbian OS Trojans such as *SymbOS/Skuller* and *SymbOS/Dampig*, which replaced system applications and antivirus files with modified ones. The focus has

recently shifted towards Android OS apps, particularly by repackaging popular games and tools [NakedSecurity, 2014], including banking apps. For example, *Android/FakeToken* trojan implements a man-in-the middle attack to forward SMS messages with mTANs (Mobile Transaction Numbers).

Zhou et al. present in [Zhou et al., 2012a] a systematic study of six popular third-party marketplaces for Android OS. Their report concludes that between 5% and 13% of all available apps online are malware using repackaging, and the most common incentive is fraud in the form of replaced in-application advertisements to re-route revenues. The study also identifies a few cases with planted backdoors and other malicious payloads.

2.3.4.2 Example: Malicious Code Transference via Network

In some cases, malware creators do not repackage an app with the full malicious code. Instead, the modified app only encloses a short piece of code that downloads and install the malicious payload once the app is installed on the device. One example of this variant—sometimes known as update attacks [Zhou and Jiang, 2012]—is *Android/DroidKungFuUpdate*. Remarkably enough, repackaged apps can enter the device without the user being aware of it. By exploiting some technical vulnerabilities and misconfigurations, some malware samples have even been able to replace another installed app by a repackaged version of the same one.

Repackaged apps often rely on obfuscation techniques to avoid detection and to make static analysis harder [Apvrille, 2011]. For example, in the case of update attacks the transferred payload is often encrypted. In other cases, encryption is applied to malicious components that are distributed together with the repackaged

app, usually as if they were class files, images or other raw resources. For instance, *Android/RootSmart* and *Android/Fjcon* use AES to hide domain names and URLs; *Android/Geinimi* conceals URLs by encrypting them with DES; and *Android/OpFake* simply makes an XOR with a predefined key.

2.3.5 Privilege Acquisition

Exploitation strategies comprise a variety of techniques used by malware to gain the privileges required to achieve its goals. We distinguish two broad classes:

- **User Manipulation**: In many cases, privileges are directly granted by users who are not aware of the potential repercussions of doing so. These strategies, which rarely involve any technical sophistication, can be surprisingly effective and very damaging. Common forms of user manipulation include:
  - Social engineering.
  - Malware and/or grayware installed by novice users who do not understand—or do not pay attention to—the permission model.
  - Repackaged applications found in alternative markets.

  As in other similar security problems in computing, these methods can be prevented by raising awareness about the dangers of malicious apps.

- **Technical Exploitation**: In other cases the malicious app can escalate by exploiting technical vulnerabilities or misconfigurations of the platform. Even though the particular technical means greatly depend on each platform, the most common current attacks include [Chin et al., 2011; Davi et al., 2011a]:

- API vulnerabilities.
- Buffer overflows.
- Code injection attacks.
- ICC vulnerabilities.
- Return-oriented Programming (ROP) and ROP without return flaws
- System vulnerabilities.
- Networking protocol flaws.
- Bootloader vulnerabilities.
- Rooted device-based vulnerabilities.

2.3.5.1 Example: Rootkits

Current smartphone platforms are becoming increasingly complex, including not only the operating system itself but also dozens of libraries that give support to the services offered by the device. Kernel-level rootkits similar to those known for traditional PCs have recently appeared with identical purposes, namely to hide the existence of malicious software from the operating system. Most rootkits infect devices via N2D vectors, but app markets—official or not—are increasingly playing a key role. For example, it is pointed out in [Zhou and Jiang, 2012] that repackaged apps that implement technical exploits to gain root access once installed in the device do exist. Such exploits are often distributed with the repackaged app or acquired from a remote server as they become available. Contrarily, other exploits involve user manipulation to acquire privilege escalation. For example, *iPhone/Mobileconfigs* [Skycure, 2014] allows an attacker to remotely hijack the device by installing malicious system-level settings into the device through social engineering.

Root exploits in iPhone are often quickly patched by Apple and it is difficult to find malware samples exploiting these vulnerabilities [Seriot, 2010]. The first exploit known for iOS was identified as early as 2007 and exploited a buffer overflow in the *libtiff* library. Other known exploits affected the SMS service—*SMS fuzzing*, presented at Black Hat USA 2009 by Miller and Mulliner—and PDF-related functionalitites—as the one used by *iPhone/JailbreakMe* to root iOS 4.3.3 and earlier versions via a web browser. Later in 2011, Miller submitted *iPhone/InstaStock* [Goodin, 2014], which, after being approved, disclosed a hidden payload endowing *InstaStock* with remotely controlled root capabilities.

Hypervisors are a common strategy to counteract rootkits. Although there are some approaches to incorporate them on smartphones, such architectures are heavyweight and not widely available yet. Bickford et al. [Bickford et al., 2010] implemented three proof-of-concept rootkits for Android. Firstly, they rootkit the GSM Linux Kernel Module (LKM) in a way that a remote attacker can listen to the victim's conversations. Secondly, they rootkit the GPS LKM so that the attacker compromises the victim's location privacy. And thirdly, they exploit a number of power-intense services so that the battery is drained in two hours. They conclude that there is currently no effective nor efficient technique to detect infection by rootkits.

2.3.6 Discussion

Table 2.2 (see page 47) shows a representative set of smartphone malware and provides, for each one of them, sought attack goals and the distribution and privilege acquisition strategies implemented. Various conclusions can be drawn:

Charact. / App	Attack Goals					Distribution / Infection						P.A.	
	Theft	Misuse	Sabotage	SPAM	Fraud	M2D	A2D	W2D	N2D	U2D	S2D	User	Exploit
FinSpy Mobile	●	□	□	–	–	–	●	●	●	●	●	●	●
Symb/Cabir	◊	◊	◊	◊	◊	–	–	–	●	–	–	●	–
Symb/Skuller	□	□	●	□	□	●	–	–	–	–	–	●	–
Symb/Yxes	●	–	●	–	–	●	–	–	–		●	●	–
Sym/ZeusMitmo	●	□	□	□	□	●	–	–	–	–	–	●	●
BB/FlexiSpy	●	–	–	–	–	●	–	–	–	–	–	●	–
BB/BBproxy	–	●	–	–	–	●	–	–	–	–	–	●	–
BB/ZeusMitmo	●	□	□	□	□	●	–	–	–	–	–	●	●
And/YZHCSMS	●	–	–	–	●	●	–	–	–	–	–	●	–
And/SpyBubble	●	–	–	–	–	●	–	–	–	–	–	●	–
And/SimChecker	●	–	–	–	–	●	–	–	–	–	–	●	–
And/BaseBridge	●	–	–	–	–	●	–	–	–	–	–	●	–
And/GinMaster	●	–	–	–	–	●	–	–	–	–	–	●	–
And/DroidKungFu	●	–	–	–	–	●	–	–	–	–	–	●	–
And/AutoSPSubs	–	–	–	–	●	●	–	–	–	–	–	●	–
And/Nickispy	●	–	–	–	–	●	–	–	–	–	–	●	–
And/Smspacem	–	●	–	●	–	●	–	–	–	–	–	●	–
And/Crusewind	●	–	–	–	–	●	–	–	–	–	–	●	–
And/Zsone	–	●	–	–	–	●	–	–	–	–	–	●	–
And/GGTracker	●	●	–	●	–	●	–	–	–	–	–	●	–
And/AdSMS	●	●	–	–	–	–	–	●	–	–	–	–	●
And/Fakeplayer	–	●	–	–	–	●	–	–	–	–	–	●	–
And/Bgserv	●	–	–	–	–	●	–	–	–	–	–	●	–
And/Lightdd	●	–	–	–	–	●	–	–	–	–	–	●	–
And/Rootcager	●	–	–	–	–	●	–	–	–	–	–	●	●
And/Opfake	–	●	–	–	–	●	●	–	–	–	–	●	–
And/OneClickFraud	–	–	–	–	●	●	–	–	–	–	–	●	–
And/FakeToken	–	–	–	–	●	●	–	–	–	–	–	●	–
iP/MogoRoad	–	–	–	–	●	–	–	●	–	–	–	–	●
iP/JailbreakMe	–	◊	–	–	–	–	–	●	–	–	–	–	●
iP/InstaStock	◊	◊	◊	◊	◊	●	–	–	–	–	–	–	●
iP/FindAndCall	●	–	–	●	–	●	–	–	–	–	–	●	–
iP/Mobileconfigs	□	□	□	□	□	–	–	●	–	●	–	●	–
iPJ/iKee.A	◊	◊	◊	◊	◊	–	–	–	●	–	–	–	●
iPJ/iKee.B	□	□	□	□	□	–	–	–	●	–	–	–	●
iPJ/Dutch 5€	–	–	–	–	●	–	–	–	●	–	–	–	●
iPJ/Privacy.A	●	–	–	–	–	–	–	–	●	–	–	–	●
WinCE/Duts.A	◊	◊	◊	◊	◊	●	–	–	–	–	–	●	–
WinCE/Fakemini	–	●	–	–	–	●	–	–	–	–	–	●	–
WinCE/Pmcryptic	–	●	–	–	–	–	–	–	–	●	–	●	–
WinCE/Terred	–	●	–	–	–	●	–	–	–	–	–	●	–
WinCE/ZeusMit.	●	□	□	□	□	●	–	–	–	–	–	●	●

Legend:

Symb: Symbian *iPJ*: Jailbroken iPhone *iP*: iPhone
And: Android *WinCE*: Windows Mobile *BB*: BlackBerry

●: The referred characteristics are applied to the application.
◊: Proof-of-concept for demonstration, novelty or amusement purposes.
□: Multi-purpose malware having multiple goals.

Table 2.2: Samples of smartphone malware for the main OS and their most relevant characteristics. Malware having multiple goals might exhibit selected characteristics depending on the specimen.

- M2D strategies clearly dominate other distribution and infection strategies. This conforms the study conducted in [Zhou and Jiang, 2012] over 1200 samples of Android OS malware, which points out that 86% of them use repackaging techniques.

- Privileges are mostly acquired by simple user manipulation, i.e., by simply asking the user to grant them to the app. This is certainly worrisome and motivates many recent works dealing with enhanced permission models and novel ways of communicating requested privileges to users. Even though repackaging is nowadays the primary entry point for malware, it is pointed out in [Zhou and Jiang, 2012] that 36.7% of studied specimes attempt to leverage technical exploits to obtain root privileges.

- In terms of behavior, malware with just one goal is rare. Most samples spy on users and steal personal data, but also attempt to commit fraud or misuse services. A possible explanation for this is the reconfigurable nature of most malware specimens through updates, as in the case of botnets. Thus, attackers basically seek to plant a basic bot engine in the device, and then to provide it with instructions and further code to perform specific tasks. Again, this conforms similar studies carried out recently. For example, in [Zhou and Jiang, 2012] it is pointed out that 90% of the samples turn the compromised device into a bot; almost half of them (45.3%) try to misuse SMS or call services to obtain financial profit; and 51.1% harvest user information. Finally, sabotage is quite unusual, with only a few examples that drain the device's battery or remove selected files.

- There are remarkable differences between Android OS and iPhone malware in

the three criteria of our taxonomy

- First, most Android OS malware is distributed by markets, notably in the form of repackaged applications. iPhone barely suffers from such infection vectors, and the majority of malware enters via web and network exploits. In part, this is a consequence of the walled-garden model of Apple's market.
- The differences in their respective permission models and the way of granting privileges also show up: while a significant fraction of Android OS malware is entitled with sufficient privileges by the user—even if it later escalates by other means—, in iPhone most specimens depend on technical exploits.
- Finally, in contrast with Android OS malware, most iPhone specimens discovered so far have been created for demonstration or amusement purposes.

A word of caution is appropriate, though: because of its openness, Android OS is the *de facto* platform-of-choice for security research in smartphones, which may have also negatively contributed to the malware phenomenon; and, furthermore, Apple follows a less communicative strategy about iPhone malware.

2.4 Malware Detection and Analysis

As detailed in the previous section, current malware pose severe threats to security models in smart devices. In this section we classify and describe the most significant advances in malware detection systems for such devices [Shahzad et al., 2012]. More

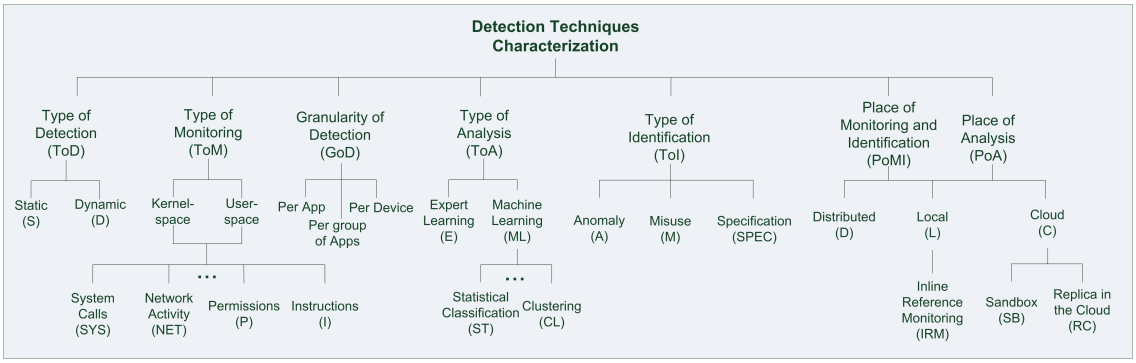


Figure 2.3: Taxonomy of malware detection techniques for smart devices.

precisely, we show how such systems build their foundations based on a variety of detection techniques. These techniques aim at identifying where and how malware manifests by constantly monitoring various device-based features. We also show how detection systems are driven by these features, as they represent the key elements for malware identification. We believe that this comprehensive study is paramount for researchers and practitioners in order to facilitate the construction of new detection systems.

2.4.1 A Taxonomy of Detection Techniques

Malware detection is a complex process pulling together monitoring, analysis and identification tasks. In order to organize and better understand current detection systems, we next propose a taxonomy based on the following seven characteristics (see Figure 2.3 for a graphical summary):

- **Type of Detection (ToD)** There are two common types of malware detection techniques according to how code is analyzed:
  - *Static analysis*: this type of technique attempts to identify malicious code by unpacking and disassembling (or decompiling) the application. This

technique is a relatively fast approach and it has been widely used in preliminary analysis to search for suspicious strings or blocks of code.

- *Dynamic analysis* techniques seek to identify malicious behaviors after deploying and executing the application on an emulator or a controlled device. These techniques require some human or automated interaction with the app, as malicious behavior is sometimes triggered only after certain events occur.

Static analysis techniques are well known in traditional malware detection and have recently gained popularity as efficient mechanisms for market protection. As a major drawback, these techniques fail to identify malicious behavior when it is obfuscated or distributed separately from the app. Contrarily, dynamic analysis are arguably more powerful in these cases. In fact, the only way of learning what the app is really doing necessarily requires to run the code and observe its actions. However, the inputs generated by most dynamic analysis tools are generally produced by using random streams of user events, which might not trigger the execution of the malicious payload, resulting in malicious apps that avoid being detected. This particular shortcoming can be tackled by modeling users' behavior and providing human-like inputs. Dynamic analysis can be used both in the cloud for market protection or directly in the device, although resource consumption is certainly a issue (see later discussion on this in Chapter 6).

- **Type of Monitoring (ToM)** Malware can be detected by analyzing various features that serve to tell apart benign from malicious activities. A monitoring system can collect *user-level*, *kernel-level*, or *hypervisor-level* activity, depending

on the type of features that will be extracted. Monitoring approaches include the collection of: (i) system calls (SYS); (ii) network activity (NET); (iii) event logs (EL); (iv) user activity; (v) instructions (I); (vi) permissions (P); or (vii) program traces (PT); to name a few. Each type of monitoring activity requires the deployment of different instruments to intercept and format the corresponding events. For instance, SYS requires the use of a system trap technique with root privileges, while NET requires capturing all packets from the network interface. Additionally, monitoring any of these features when the app is run in an hypervisor requires the introspection of a virtual environment.

Monitoring can be potentially expensive in terms of resource consumption, particularly if a large number of events is collected directly over the platform being monitored. As far as we are aware, no power consumption analysis has been carried out yet, but practical experience suggests that intensive monitoring is prohibitive for current smart devices.

- **Granularity of Detection (GoD)** A point related to the ToM discussed above is how collected data is filtered in order to select the detection scope. Monitoring can be carried out at different levels:

  - *Per App*: features related to a specific application are monitored and analyzed independently from other apps in the system. This type of feature classification presents good performance when malware is a stand-alone application.

  - *Per group of apps*: in this case, data from a collection of applications is gathered and analyzed. This is potentially useful when malware's goals are achieved in a distributed way by several collaborating apps.

  - *Per device*: detecting certain types of malware, such as for example rootkits, requires a more general detection approach focused on monitoring the device itself rather than particular apps executed on it.

- **Type of Analysis (ToA)** The monitored information is subsequently analyzed to extract evidence on the presence of malware. Such analysis can be carried out by a human expert (E), although this possibility is becoming increasingly unaffordable, at least without the support of automated analysis tools. There are several types of techniques for analyzing data obtained after monitoring, including: Clustering (CL), Support Vector Machines (SVM), Self-Organizing Maps (SOM), other general Machine Learning (ML) algorithms, Control Flow Graphs (CFG), Data Flow Graphs (DFG), Program Dependency Graphs (PDG), etc.

- **Type of Identification (ToI)** Depending on the type of identification carried out, detection systems can be classified as either *anomaly*-based (A), *misuse*-based (M), or *specification*-based (SPEC) system. This feature refers to the principle guiding the identification of malicious activities and follows the same ideas explored in Intrusion Detection Systems [Estévez-Tapiador et al., 2004; Garcia-Teodoro et al., 2009].

  - *Anomaly-based* identification attempt to model the "normal" behavior of the monitored system, classifying as anomalous any other behavior reported. Anomaly detection techniques have the potential to detect previously unseen malware. However, they generally present a high rate of false positives, i.e., they are prone to detect rare legitimate behaviors as malicious.

  - *Misuse-based* identification—also known as signature-based—aims at identifying known malicious activity by means of predefined patterns of signatures. Thus, only "malicious" behaviors are modeled here. The main benefit of misuse detection lies in its accuracy detecting well-known attacks. Generally, for each known malicious behavior, misuse systems are equipped with one or more signatures. In this regard, maintaining an up-to-date database with a massive amount of signatures poses a major challenge. Furthermore, resource-constrained devices are not capable of processing big amount of signatures.
  - *Specification-based* identification works on the basis of predefined authorized behaviors (specifications) and assumes that any activity deviating from them violates the system policy and, therefore, is malicious.

- **Place of Monitoring and Identification (PoMI)** Monitoring, analysis, and identification techniques are generally resource-intensive tasks that cannot be afforded in battery-constrained devices. As a consequence, in recent years it has been proposed to externalize many of such tasks to more powerful platforms, even though some processing still needs to be taking place in the device. We distinguish three main classes of detection schemes according to where monitoring and identification takes place:

  - *In the device*: both monitoring and identification are placed locally in the device. This requires very lightweight approaches and their scope may be quite limited. There are two types of local monitoring or identification techniques according to where the monitoring is taking place:

    * Local out-line (L): this type of technique aims at monitoring the de-

vice by installing itself in one of the lower layers of the device's architecture, and generally require root privileges.

* Local in-line, also known as *Inline Reference Monitor (IRM)*: this type of technique rewrites untrusted applications so that the monitoring code is embedded into the app, and does not require root privileges.

– *Distributed (D)* among other devices. Performs any monitoring, analysis or identification task in a cooperative way among different trusted devices.

– *In the cloud (C)*. Uses virtual environments for running several devices on a single server machine without reducing the battery life.

* Sandbox (SB): uses a tightly controlled set of resources for running dynamic analysis over target apps.

* Replica in the cloud (RC): uses remote security servers for hosting exact replicas of the device. Monitoring and identification techniques that are placed on the replicas require complex synchronization systems to ensure that the replica is at all times identical to the actual device, as well as collaboration with the service provider (e.g., the internet provider for general purpose devices or phone provider for smartphones).

- **Place of Analysis (PoA)** Finally, depending on where the analysis component is placed—i.e., locally or in the cloud—the approach used poses different challenges. On one hand, *cloud-based* approaches require local preprocessing of the monitored traces, transmitting them to the cloud, and waiting for the results. Finally, results may be included for further identification of malware. On the other hand, *local* approaches might accelerate the delay in obtaining the

response, especially when traces are too big and/or the connection is very slow.

2.4.2 Monitorable Features in Smart Devices

According to the monitoring approaches discussed above, we next identify and classify a number of device-based features that can provide evidence of malware activities. We subsequently explore how the behavior of some representative classes of malicious activities manifest in subsets of these features. Specifically, we analyze those features against: (i) *botnets*–like malware, (ii) Denial of Service (*DoS*) attacks, (iii) technical exploitations, i.e., *SMS–of–death*, (iv) user manipulation such as *Phishing* or *Pharming*, (v) information theft via *monitoring*, and (vi) service misuse such as *SMS* or (Quick Response) *QR* codes. A summary of this taxonomy—excluding the full list of features for each class—is given in Figure 2.4.

- **Hardware**: this kind of features identify the state of the hardware (HW) components of the device. We group HW features in three subclasses: (i) *battery*, (ii) *input/output HW*, and (iii) *device info*. Table 2.3 provides a detailed list of features for each subclass. The state of the battery or the access to the unique device identifier can be used to detect a specific type of malware. For instance, some botnets check first that the battery is charging before performing heavy operations. Another example of the use of HW-based features for malicious purposes is access to the IMEI of a smartphone with the goal of exfiltrating it.

- **Communications**: communications represent an essential infection vector in smartphones. They include the following features: (i) phone and internet *calls*, (ii) phone and internet *messaging*, and (iii) *network* usage (data other than calls and messaging), as identified in Table 2.4.

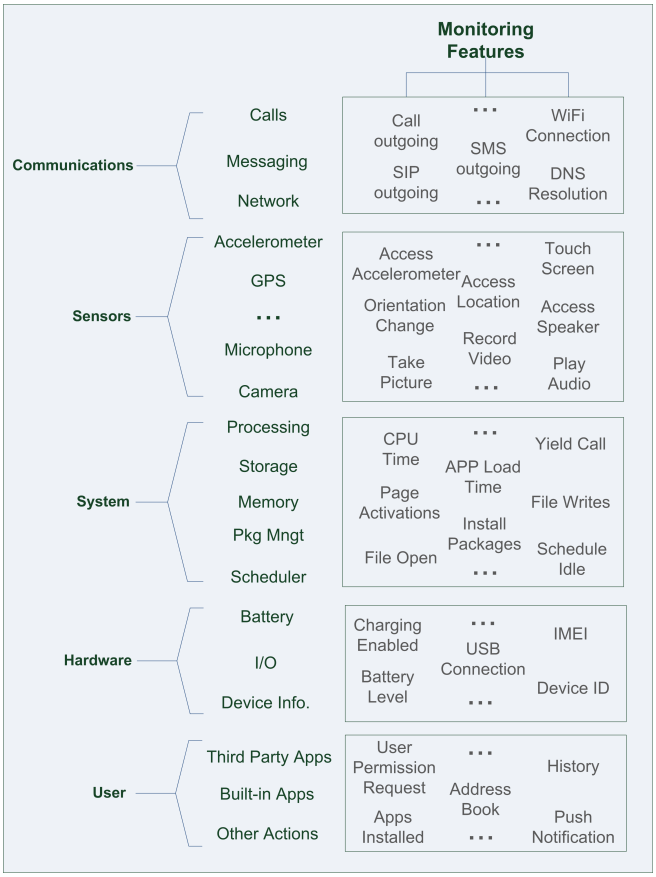


Figure 2.4: Taxonomy of monitorable features for smart devices.

- **Sensors**: on-platform sensors allow the device to interpret the physical context of a user [Knappmeyer et al., 2013]. Currently the most common sensors are: (i) *accelerometer*, (ii) *GPS*, (iii) *compass*, (vi) *gyroscope*, (v) *microphone*, (vi) *touch sensors*, (vii) *speakers*, and (viii) *camera*, as illustrated in Table 2.5. Access to sensors can be monitored to identify malicious use. For instance, profiling malware will typically access the user's current location. Thus, if an application is constantly accessing the GPS and sending this information through the network, it could be an indication of malicious—or, at least, potentially dangerous—usage.

- **System**: access to system resources can be used to identify malicious behaviors

Features	Attacks	Botnet	DoS & DDoS	SMS–of–death	Phishing	Pharming	Monitoring	Misuse SMS service	Malicious QRCode
Battery	Charging_Enabled	•	–	–	–	–	–	–	–
	Battery_Voltage	•	•	–	–	–	–	–	–
	Battery_Current	•	•	–	–	–	–	–	–
	Battery_Temp	•	•	–	–	–	–	–	–
	Battery_Level_Change	•	•	–	–	–	–	–	–
I/O	LED	–	–	–	–	–	–	–	–
	USB_Connection	–	–	–	–	–	–	–	–
	Coverage_Range	–	–	–	–	–	–	–	–
	Press_Key	–	–	–	–	–	•	•	–
Device Info.	IMEI	–	–	–	–	–	•	–	–
	Device_Id	–	–	–	–	–	•	–	–
	SIM_Card	–	–	–	–	–	•	–	–
	Phone_State	–	–	–	–	–	•	–	–
	UID_Access	–	–	–	–	–	•	–	–
	UID_Removal	–	–	–	–	–	•	–	–

Table 2.3: Monitorable HARDWARE features and examples of attacks that could affect them.

by monitoring: (i) *processes*, (ii) *storage*, (iii) *memory*, (iv) *package management*, and (v) *scheduler*, as identified in Table 2.6.

- **User**: there are a number of features that generally involve user interaction and that could also provide evidence of malicious behavior. We identify (i) *user-permissions* frequency requests (applications can be classified into categories by monitoring the frequency at which they request permissions [Rassameeroj and Tanahashi, 2011]), (ii) *third-party apps*, (iii) *built-in apps*, and (iv) *other actions*, as detailed in Table 2.7.

2.4.2.1 Discussion

Malicious apps—as any other app—rely on the device's system and sensors to achieve their goals. Different components of the device are therefore interrogated by the mal-

Features		Attacks	Botnet	DoS & DDoS	SMS–of–death	Phishing	Pharming	Monitoring	Misuse SMS service	Malicious QRCode
Calls	Phone	Phone_Outgoing	–	●	–	–	–	●	–	–
		Phone_Incoming	–	●	–	–	–	●	–	–
		Phone_Missed	–	●	–	–	–	●	–	–
		Phone_Privileged	–	●	–	–	–	●	–	–
	Internet	SIP_Incoming	–	●	–	–	–	–	–	–
		SIP_Outgoing	–	●	–	–	–	●	–	–
Msg.	Phone	SMS_Incoming	●	●	●	–	–	●	–	–
		SMS_Outgoing	●	●	–	–	–	●	–	–
		SMS_Read	●	●	–	●	–	●	–	–
		SMS_Privileged	–	●	–	–	–	●	●	–
		MMS_Incoming	●	●	●	–	–	●	–	–
		MMS_Outgoing	●	●	–	–	–	●	–	–
		MMS_Read	●	●	–	●	–	●	–	–
		MMS_Privilege	–	●	–	–	–	●	●	–
	Internet	XMPP_Incoming	●	–	–	–	–	●	–	–
		XMPP_Outgoing	●	●	–	–	–	●	–	–
Net.	Byte	WiFi_TX_Bytes	●	–	–	–	●	●	–	–
		Phone_TX_Bytes	●	●	–	–	–	●	–	–
		Bluetooth_TX_Bytes	●	●	–	–	–	●	–	–
		WiFi_RX_Bytes	●	●	–	●	–	●	–	–
		Phone_RX_Bytes	●	●	–	–	–	●	–	–
		Bluetooth_RX_Bytes	●	●	–	–	–	●	–	–
	Packets	WiFi_TX_Pckts	●	●	–	–	●	●	–	–
		Phone_TX_Pckts	●	●	–	–	–	●	–	–
		Bluetooth_TX_Pckts	●	●	–	–	–	●	–	–
		WiFi_RX_Pckts	●	●	–	●	–	●	–	–
		Phone_RX_Pckts	●	●	–	–	–	●	–	–
		Bluetooth_RX_Pckts	●	●	–	–	–	●	–	–
	Connections	WiFi_CX	●	●	–	●	●	●	–	–
		Phone_CX	●	●	–	●	●	●	–	–
		Bluetooth_CX	●	●	–	●	●	●	–	–
		DNS_Resoluc.	●	●	–	●	●	●	–	–

Table 2.4: Monitorable COMMUNICATIONS features and examples of attacks that could affect them.

ware to operate. For instance, the behavior of botnets is deeply related to almost any kind of communication feature as all bots rely on a C&C back-end. Additionally, they could also require some system interactions in order to store and update themselves. However, they are not likely to access any sensor—unless the master commands it

Features	Attacks	Botnet	DoS & DDoS	SMS–of–death	Phishing	Pharming	Monitoring	Misuse SMS service	Malicious QRCode
Accelerometer	Access_Accelerometer	–	–	–	–	–	•	–	–
	Current_Roll_Pitch_Yaw	–	–	–	–	–	•	–	–
	Orientation_Changing	–	–	–	–	–	•	–	–
GPS	Access_Location	–	–	–	–	–	•	–	–
	Current_Location	–	–	–	–	–	•	–	–
	Location_Changing	–	–	–	–	–	•	–	–
Compass	Access_Compass	–	–	–	–	–	•	–	–
	Current_Cardinal_Orientation	–	–	–	–	–	•	–	–
	Cardinal_Orientation_Changing	–	–	–	–	–	•	–	–
Gyroscope	Access_Gyroscope	–	–	–	–	–	•	–	–
	Current_Angular_Moment	–	–	–	–	–	•	–	–
	Angular_Moment_Changing	–	–	–	–	–	•	–	–
Microphone	Record_Audio	–	–	–	–	–	•	–	–
	Access_Audio	–	–	–	–	–	•	–	–
Touch	Touch_Screen_Preasure	–	–	–	–	–	•	–	–
	Touch_Screen_Area	–	–	–	–	–	•	–	–
Speaker	Access_Speakers	–	–	–	–	–	•	–	–
	Play_Audio	–	–	–	–	–	•	–	–
Camera	Take_Picture	–	–	–	–	–	•	–	•
	Access_Picture	–	–	–	–	–	•	–	•
	Record_Video	–	–	–	–	–	•	–	–
	Access_Video	–	–	–	–	–	•	–	–
	Calculate_Depth (RGDB)	–	–	–	–	–	•	–	–

Table 2.5: Monitorable SENSORS features and examples of attacks that could affect them.

through a remotely transmitted payload. Another interesting example is given by fraud attacks such as *Phishing* or *Pharming*. In these cases, the malware is likely to use network connections in order to get to the victim, access to SMS messages to steal, for example, One Time Passwords (OTPs), or change the DNS resolution of the device, but it will definitely not access sensors.

Accessing those components in a stealthy manner is still, to the best of our knowledge, a limitation for attackers. Nevertheless, there are some technical exploitation vectors that allow a malware to root the device, which could thwart detection at

Features	Attacks	Botnet	DoS & DDoS	SMS–of–death	Phishing	Pharming	Monitoring	Misuse SMS service	Malicious QRCode
Processing	CPU_Time	–	•	•	–	–	–	–	–
	Runnable_Entities	–	•	–	–	–	–	–	–
	Context_Switching	–	–	–	–	–	–	–	–
	Wakelocks	–	–	–	–	–	–	–	–
	Processes_Changing	–	•	–	–	–	–	–	–
Storage	File_Open	•	–	–	–	–	–	–	–
	File_Reads	•	–	–	–	–	–	–	–
	File_Writes	•	–	–	–	–	–	–	–
	File_Read_Bytes	•	–	–	–	–	–	–	–
	File_Write_Bytes	•	–	–	–	–	–	–	–
Memory	Dirty_Pages	–	–	–	–	–	–	–	–
	Active_Pages	–	–	–	–	–	–	–	–
	Anonymous_Pages	–	–	–	–	–	–	–	–
	Page_Activations	–	–	–	–	–	–	–	–
	Page_Desactivations	–	–	–	–	–	–	–	–
	Page_Faults	–	–	–	–	–	–	–	–
	DMA_Allocations	–	–	–	–	–	–	–	–
	Garbage_Collections	–	–	–	–	–	–	–	–
	Page_Frees	–	–	–	–	–	–	–	–
	Inactive_Pages	–	–	–	–	–	–	–	–
	File_Pages	–	–	–	–	–	–	–	–
	Mapped_Pages	–	–	–	–	–	–	–	–
	Writeback_Pages	–	–	–	–	–	–	–	–
Pkg Mgmt	App_Load_Time	•	–	–	–	–	–	–	–
	Install_Packages	•	–	–	–	–	–	–	–
	Delete_Packages	•	–	–	–	–	–	–	–
	Change_Package	•	–	–	–	–	–	–	–
	Restart_Package	•	–	–	–	–	–	–	–
	Master_Clear	•	–	–	–	–	–	–	–
Scheduler	Yield_Calls	–	–	–	–	–	–	–	–
	Schedule_Idle	–	–	–	–	–	–	–	–
	Running_Jiffies	–	–	–	–	–	–	–	–
	Waiting_Jiffies	–	–	–	–	–	–	–	–

Table 2.6: Monitorable SYSTEM features and examples of attacks that could affect them.

some levels. In those cases, access to hypervisor-level monitoring is paramount to identifying such cases.

Tables 2.3 through 2.7 present various examples of malicious activities and the

Features	Attacks	Botnet	DoS & DDoS	SMS–of–death	Phishing	Pharming	Monitoring	Misuse SMS service	Malicious QRCode
User–permissions	#_requests	•	•	•	•	•	•	•	•
Third Party Apps	Apps_Installed	•	–	–	–	–	–	–	–
	Apps_Usage	•	–	–	–	–	–	–	–
	Apps_Delete	•	–	–	–	–	–	–	–
Built–in Apps	Address_Book	–	–	–	–	–	•	–	–
	History	–	–	–	–	–	•	–	–
	Bookmarks	–	–	–	–	–	•	–	–
	Calendar	–	–	–	–	–	•	–	–
	Feeds	–	–	–	–	–	•	–	–
	Email	–	–	–	–	–	•	–	–
Other Actions	Push_Notifications	–	–	–	–	–	•	–	–
	Unlock	•	•	•	•	•	•	•	–

Table 2.7: Monitorable USER features and examples of attacks that could affect them.

features that would likely allow a detection system to identify them. The mapping between the monitorable features and the attacks has been extracted analytically based on the criterion and the expertise of the authors. Based on this, several conclusions can be drawn:

- Monitoring can be a very heavy consuming task. Thus, identifying a monitoring strategy as well as an appropriate *type* of features is crucial to reduce workload and improve detection efficacy. For instance, if a user is interested in using his device in a Bring-Your-Own-Device (BYOD) context, avoiding exfiltration of sensitive information may be critical, and therefore monitoring only some specific features would be a good strategy.

- From all eight cases studied, the most relevant group of features affects communications (Table 2.4). In this regard, it is also interesting to identify adaptive monitoring strategies based on the appropriate *amount* of features. Thus, if

a detection system can likely identify the most popular malware by only monitoring, say, 40% of the features, then monitoring the remaining ones can be eventually switched off, e.g., when the battery is lower than a given threshold.

Finally, we emphasize that the list of detection features presented in Tables 2.3 through 2.7 are only an excerpt of all those that can be used by a detection system. However, . In general, each type of device will offer a more or less exhaustive list of available features for each category given above.

2.4.3 Overview of Detection Systems

In the last few years several works have been proposed to detect malware on smart devices—mostly smartphones and, more specifically, for Android OS platforms. We have classified the 20 most representative detection systems according to the taxonomy provided above. The result, shown in Table 2.10, summarizes current research directions.

Even though all detection systems are strongly interrelated, some general characteristics are evident. For example, while some techniques are more versatile and, therefore, are used more often, others are used mainly for certain detection systems. Thus, both static and dynamic analysis are used for device and market protection. However, it is more frequent to use dynamic analysis for device-oriented detection and static analysis for market protection. Despite this, dynamic analysis is becoming an important technique for market detection as well, as new paradigms based on Security-as-a-Service, such as *Replicas in the Cloud*, are gaining popularity.

For the sake of organization, in the remaining of this section we describe current research proposals grouped into three main categories:

i) Device monitoring systems.

ii) Automatic app-review systems for market protection.

iii) Attack-specific malware identification systems (both for user and market protection).

2.4.4 Device-based Monitoring Systems

Device-based malware detection systems have received much attention lately. They mostly use dynamic analysis techniques, although some combine them with static analysis to improve the detection strategy. In this regard, both anomaly and misuse detectors are proposed.

2.4.4.1 Anomaly Detectors

Schmidt et al. [Schmidt, 2011] leverage both static and dynamic analysis for detecting malware in Symbian OS and Android OS devices. On the one hand, function calls are first extracted, and monitored data is then analyzed using decision trees. Classifiers are trained to recognize normal and malicious apps. On the other hand, an anomaly-based malware detection is used for dynamic analysis. Features such as free RAM memory, CPU usage, SMS count, etc. are monitored for further analyzing behavior. Analysis is done in the cloud using machine learning algorithms such as Artificial Immune Systems (AIS), Self-Organizing Maps (SOM), Support Vector Machines (SVM), and Tree Kernels.

A somewhat similar approach is Andromaly [Shabtai et al., 2012], which uses dynamic analysis for periodically monitoring a number of features and machine learning

LEGEND

Platform
- **And:** Android
- **Win:** Windows
- **Sym:** Symbian

Type of Detection (ToD)
- **S:** Static
- **D:** Dynamic

Other
- **∅:** Unavailable

Type of Monitoring (ToM)
- **SYS:** System calls
- **NET:** Network
- **EL:** Event Log
- **I:** Instructions
- **P:** Permissions
- **PT:** Program Traces
- **PCB:** Process Control Block
- **API:** API Calls
- **K:** Kernel-level
- **U:** User-level

Type of Analysis (ToA)
- **E:** Expert
- **ML:** Machine Learning
- **CL:** Clustering
- **DG:** Dependency Graphs
- **ST:** Statistical
- **PRO:** Probabilistic Models

Type of Identification (ToI)
- **A:** Anomaly
- **M:** Misuse
- **SPEC:** Specification

Place of Monitoring and Identification (PoMI) and Place of Analysis (PoA)
- **L:** Local Outline
- **IRM:** Local Inline (IRM)
- **C:** Cloud
- **DB:** Distributed
- **HP:** Honeypot
- **RC:** Replica in the Cloud
- **SB:** Sandbox
- **H:** Hybrid

		Detection Approach									
	Plat.	ToD	ToM	ToA	ToI	PoMI	PoA	Consump.	Features	Attack	Observations
App Profiler (2013) [Rosen et al., 2013]	And	S, D	API, PT	E	M	L	L, C	Not available	Permissions, and API Calls	Privacy leakage	API calls are analyzed statically using signatures and apps are traced dynamically through tainting analysis
Apps Playground (2013) [Rastogi et al., 2013a]	And	D	SYS, PT	∅	∅	C	C	Not applicable	Taint tracing, SYS call, etc.	Any kind	Heuristic-based UI interaction based on contextual exploration
Secloud (2013) [Zonouz et al., 2013]	And	*	*	*	*	RC	C	Device consumption not available	Any kind	Any kind	Detection techniques: AV scanning, file integrity checking, SYS call monitoring, or network intrusion detection and response
TStruct Droid (2013) [Shahzad et al., 2013]	And	D	PCB	ST, ML	A	L	L	Performance degradation of 3.73% on average	Frequencies of 99 preliminary parameters.	Any kind	Type of analysis: theoretic analysis, time–series feature logging, segmentation and freq. component analysis of data, and ML classifier
Andromaly (2012) [Shabtai et al., 2012]	And	D	*	ML	A	L	L	≈ 8.8% RAM, 5.52% CPU, and 10% Battery (unclear)	Detection Method: monitorization of a subset of 88 initial features	Any kind of anomaly	Training Method: Classification with labelled data. Experimental evaluation
AppGuard (2012) [Backes et al., 2012]	And	D	PT	∅	M	IRM	C	Not available	Program traces and generated events	Privacy leakage and user–level misuse	Analysis is done offline, prior to repackaging the app, i.e., in the cloud

Table 2.8: Malware detection systems (I/III).

	Plat.	ToD	ToM	ToA	ToI	PoM	PoA	Consump.	Features	Attack	Observations
Elish et al. (2013) [Elish et al., 2013]	And	S	I	DG	∅	C	C	Not applicable	Data event–specific control	Component hijacking for information leakage and unauthorized access	Uses DDG to track the user's private information
DroidScope (2012) [Yan and Yin, 2012]	And	D	*	∅	∅	SB	C	Not applicable	Any kind	Any kind	ToM: Syscalls, etc. Ad–hoc plugins for monitoring features and analyzing data (authors provide several proof of concepts, e.g.: tainting)
MADAM (2012) [Dini et al., 2012]	And	D	K, U	ML	A	L	L	Overhead of 3% memory, 7% CPU and 5 % battery	K: SYS, proc., memory, CPU usage. U: user–state, key strokes, called numbers, SMS, NET	Any kind of anomaly	K-NN (with K=1) for classification. 10 malicious apps and 50 benign. 93% detection rate and 5% FP
Peng et al. (2012) [Peng et al., 2012]	And	S	P	PRO	N, A	C	C	Not applicable	Permissions	Effectiveness of apps permissions	
RiskRanker (2012) [Grace et al., 2012b]	And	S	I, P, API	DG	M	C	C	Not applicable	Vulnerability signatures, permissions, API calls: crypto, dynamic code, IPC, and JNI, etc.	Any kind	Checks a pre-defined set of malicious operations (e.g.: known exploits) to rate the severity of stealthy applications
SmartDroid (2012) [Zheng et al., 2012]	And	H	*	*	*	SB	SB	Unavailable	Any	UI–based obfuscation	Improved detection by generating UI-based trigger conditions. Any kind of detection system might be plunged, but no further details are given
Schmidt et al. (2011) [Schmidt, 2011]	Sym, And	S, D	SYS	CL	A	L	C, DB	Not available	Free RAM, User Inactivity, Process count, CPU usage, SMS, etc.	Any kind of anomaly	Training method: SVM–light and user's statistical data
Crowdroid (2011) [Burguera et al., 2011]	And	D	SYS	CL	A	L	C	Not available	System calls per application	Any kind of anomaly	Training Method: Clustering with k–means: i) malware, and ii) goodware. Evaluation: Experimental and wild malware

Table 2.9: Malware detection systems (II/III).

	Plat.	ToD	ToM	ToA	ToI	PoM	PoA	Consumption	Features	Attack	Observations
Woodpecker (2012) [Grace et al., 2012a]	And	S	I, P	DG	∅	C	C	Time consuming analysis: 1 hour per phone image	Executing paths and 13 representative privileged permissions	Capability leaks and confused deputy attacks	Uses CFG for detecting explicit capability leakages and permissions for implicit
CHEX (2012) [Lu et al., 2012]	And	S	I	DG	∅	C	C	Not applicable	User's data	Component hijacking for information leakage and unauthorized access	Uses system dependence graphs to track the user's private information
AASandbox (2010) [Blasing et al., 2010]	And	D	*	CL	M	SB	C	Not applicable	Not available	Any kind	Training method: Unspecified type of clustering. Evaluation: Self–written malware
Paranoid Android (2010) [Portokalidis et al., 2010]	And	D	*	*	*	RC	C	Discussed. Apparently larger than expected	Not available	Any kind	Training method: Dynamic analysis and AV Analysis. Evaluation: Not performed
TaintDroid (2010) [Enck et al., 2010]	And	D	PT	E	M	L	L	Uses 14% CPU and 4.4% memory overhead. Power consumption not available	Variables, methods, file, and message	Explicit information flow leakage	Type of monitoring: label-based tracking of variables, methods, files and IPC via dynamic tainting, and enforced by the user. Tainted variables are propagated according to data flow rules
Kim et al. (2008) [Kim et al., 2008]	Win	D	HW	ST	M	L	L, C	Not available	Energy consumption	Energy-depletion attacks	The consumption is monitored using physical hardware (HW) and the analysis is done either at the phone or at the server (no performance comparison is provided)

Table 2.10: Malware detection systems (III/III).

anomaly-based detectors for classifying apps as goodware or malware. In Andromaly, however, classification is done locally in the device. The scheme monitors various system features such as CPU consumption, number of network packages, number of running processes and battery level. Redundant features are first eliminated using three feature selection algorithms: Chi-Square, Fisher Score, and Information Gain.

Furthermore, collected observations are classified using K-Means, Logistic Regression, Histograms, Decision Trees, Bayesian Networks and Naive Bayes. Evaluation was performed testing a small number of self-implemented malware samples, and results show a detection rate accuracy ranging from 44% to 100%. More precisely, they show that Fisher Score with 10 top features selected, and using Naive Bayes and Logistic Regression, perform better than the other classifiers. Although no real malware is studied, their experiments help to understand which machine learning algorithms are superior as well as their degradation. In fact, their experiments show a 10% of performance degradation in the worst scenario, i.e., 8 different classifiers with 30 features. However, it is not clear how this performance has been measured and whether the consumption exhibited is in the same conditions with the malware detector or without it.

Similarly to Andromaly [Shabtai et al., 2012], MADAM [Dini et al., 2012] uses dynamic analysis for periodically monitoring a number of features, and machine learning anomaly detectors for classifying goodware and malware, locally in the device. However, MADAM is evaluated using real malware samples, and consequently needs a higher number of features to model user behavior. Furthermore, collected observations are classified using K-Nearest Neighbor (K-NN) with $K = 1$ (1-NN). The evaluation was carried out with more than 50 goodware applications and 10 malware samples along with several user behaviors, improving the detection accuracy (93%) with respect to the same classifier used in Andromaly [Shabtai et al., 2012]. The results show an average number of number of 5 false positives per day. The reported performance overhead is 3% of memory consumption, 7% of CPU overhead and 5% of battery.

More recently, TStructDroid [Shahzad et al., 2013] presents a real-time malware

detection system for Android OS devices. The proposed system monitors Process Control Blocks (PCB) and uses theoretical analysis, time-series feature logging, segmentation and frequency component analysis of data, and a learned classifier to analyze monitored data. Evaluation shows a 98% accuracy and less than 1% false alarm rate, together with a 3.73% of performance degradation.

Finally, Crowdroid [Burguera et al., 2011] is another anomaly-based malware detection system for Android OS devices. The main difference with Andromaly [Shabtai et al., 2012] and MADAM [Dini et al., 2012] is that authors analyze the monitored featured in the cloud, whereas the other two approaches train their classifiers locally in the device. Collected observations are classified using K-Means. Evaluation was also carried out using a self-implemented set of malware samples, showing a detection rate of 100%. Additionally, they also test their system with two malware instances observed in the wild, showing a detection rate of 85% and 100% respectively. A key limitation in their study is that they assume that outsourcing the analysis should present a lower battery degradation than approaches that classify locally. However, we consider that this assumption has to be formally proven as some detection approaches are quite lightweight and might consume less than continuously transmitting all traces through the network.

2.4.4.2 Misuse Detection

AppGuard [Backes et al., 2012] is a malware prevention system for Android OS in which the monitoring system is placed inline (IRM) with the application. Applications are manipulated using the repackaging technique, and the monitoring system is, therefore, inserted inside the applications. Applications can thus trace themselves

and a number of security policies can be defined to enforce system permissions at run-time. Evaluation was performed using 13 apps, each of which was inlined with 9 policies. One noteworthy characteristic is that inlined apps incur a negligible increment in their size.

Reported experiments in [Backes et al., 2012] also compare the execution of three function calls in both the original and the inlined app (the latter with no policies set), showing a degradation of 5.0%, 6.2%, and 1.0% of overhead respectively. In this regard, we consider that the three micro-benchmarks used are not conclusive due to their simplicity. Additionally, we consider that these results cannot be compared with Andromaly as they were not tested under the same conditions.

2.4.4.3 Replicas in the Cloud

Approaches such as Paranoid Android [Portokalidis et al., 2010] or Secloud [Zonouz et al., 2013] have focused on performing malware detection tasks over synchronized replicas of the device maintained in the cloud. Thus, all security monitoring, analysis and identification tasks can be done in an environment not subject to battery constraints. Additionally, multiple detection techniques can be applied simultaneously, as several replicas can be run at the same time.

The proposed systems introduce several attack detection mechanisms for dynamic analysis in the replicas such as AV scanners and tainting analysis. However, Secloud [Zonouz et al., 2013] extends those mechanisms and deploys a number of response and prevention techniques, including file removal, process termination, periodic backups, network filtering, and device quarantining.

Experiments on Paranoid Android [Portokalidis et al., 2010] show that synchro-

nizing the device with the replicas does not introduce more than 2KB/s and 64B/s of trace data for high-load and idle operation environments, respectively. This performance, however, cannot be compared with Secloud [Zonouz et al., 2013], as for the latter no information about the consumption of the device being replicated is provided.

2.4.5 Market Protection

Most of the aforementioned techniques are typically designed to monitor physical devices, although they can also be used in virtual environments for market protection. Using specific monitoring techniques for virtual environments can bring about a number of benefits, such as (i) performing a resource-intensive security analysis, (ii) enabling virtual machine introspection [Garfinkel et al., 2003] to intercept OS-level semantics, or (iii) enabling the possibility of hosting exact replicas of the device in the cloud (e.g.: CloneCloud [Chun et al., 2011], and ThinkAir [Kosta et al., 2012]) as mentioned before.

2.4.5.1 Sandboxing

Several approaches have been proposed for malware detection in the form of sandboxes. For example, AASandbox [Blasing et al., 2010] is an Android OS analysis sandbox for both static and dynamic analysis. AASandbox uses an android emulator, pre-loaded with a SYS call monitoring service.

DroidScope [Yan and Yin, 2012] is another sandbox for Android OS based on virtualization. It allows to monitor app features at the three layers of Android OS's architecture, i.e., hardware, OS, and Dalvik Virtual Machine. Different types of

monitoring can be enabled by developing custom plugins over DroidScope. In this regard, the authors include (i) a collector for native and Dalvik instructions traces, (ii) a profiler for API-level activity, and (iii) a tracking system for information leakage using taint analysis.

2.4.5.2 Smart Interaction

Sandbox analysis poses a limitation when interacting with samples in an automated way, due to the fact that some malicious apps hide their malicious activity through the User Interface (UI). In this regard, SmartDroid [Zheng et al., 2012] presents an hybrid static and dynamic detection method to reveal UI-based trigger conditions in Android OS. While static analysis is used to generate Activity and Function Call Graphs (ACG and FCG, respectively), dynamic analysis is used to explore such paths.

AppsPlayground [Rastogi et al., 2013a] presents a similar approach combining detection techniques (ranging from taint tracing to SYS call monitoring) along with automatic exploration strategies. The proposed framework uses heuristics to guide the UI inputs, avoiding redundant explorations and using contextual information to fill editable text boxes.

2.4.5.3 Risk Analysis

Risk analysis techniques are emerging as a mechanism to palliate the ineffective way in which permissions are used to communicate potential threats to the user [Felt et al., 2011c]. Here, Grace et al. propose the use of static assessment metrics to measure dangerous behaviors in Android OS called RiskRanker [Grace et al., 2012b]. Their proposal focuses on conducting a scalable, efficient and accurate proof-of-

concept rather than leveraging on sophistication. Contrary, Peng et al. [Peng et al., 2012] propose the use use probabilistic generative models for risk ranking and scoring schemes. More precisely, they evaluate a range of models starting from simple Basic Naive Bayes (BNB) to advanced hierarchical mixture models, showing that these models offer a promising mechanism for risk scoring.

2.4.5.4 Similarity detection

Researchers have explored different ways to detect repackaging in markets by detecting similarity dependencies among population of applications. While early approaches use syntactic analysis such as string-based matching [Desnos, 2012], recently approaches elaborate on semantic analysis [Crussell et al., 2012], e.g., PDG, as it is resilient to code obfuscation. However, semantic analysis is generally more expensive than syntactic analysis.

A different approach is presented in [Desnos, 2012], where several compression algorithms are used to compute normalized information distances between two applications based on Kolmogorov complexity measurement. Their algorithm first identifies which methods are identical and calculates the similarity of the reminder methods using Normalized Compression Distances (NCD). In order to reduce complexity, the authors use a representation of each method based on structured control flow signatures [Cesare and Xiang, 2010]. Finally, authors apply Longest Common Subsequence (LCS) algorithm to identify differences between similar elements.

Zhou et al. [Zhou et al., 2012a] propose a system called DroidMOSS for detecting repackaged applications based on a fuzzy hashing technique. Distinguishing features are first extracted in the form of fingerprints, and then compared with those from

other applications in order to identify similarities. These features are computed by applying traditional hash functions to pieces of code of variable size. The size of the pieces is bounded by smaller chunks of fixed size called reset points. A chunk is considered a reset point when the resulting hash is a prime number. Then, the edit distance is calculated between two applications by comparing their fingerprints on identical matching-basis. More recently, authors have extended their work in [Zhou et al., 2013]. While their former work is designed to detect repackaging in unofficial markets, the latter is capable of detecting repackaging among apps in the same market.

Authors in [Hanna et al., 2013] present Juxtapp, a system for detecting app similarity. They propose an optimization over the representation of the applications as an alternative to k-grams based on feature hashing and then use hierarchical clustering to classify similar applications.

Authors in [Crussell et al., 2012] present DNADroid, a system for detecting cloned applications based on dependency graphs between methods. PDG is used to detect semantic similarities by comparing graph isomorphism. Prior to similarity detection, authors group applications based on meta-information retrieved from each application, and they use several filters to enhance efficiency. Although their experiments show better results than similar approaches such as [Desnos, 2012], the scheme is less efficient in terms of performance. In fact, their experimental testbed is deployed in a small cluster composed of one server and three desktop computers over *Hadoop*. Even there, the analysis rate is 0.7 applications per minute.

2.4.6 Attack-specific Malware Identification Systems

The majority of the approaches described above focus on general detectors using either anomaly or misuse detection for both static and dynamic analysis. However, due to the diversity of malware goals and incentives, other schemes are narrowing the complexity towards detecting specific classes of malware, such as privileged escalation, battery-depletion attacks, or money stealing.

2.4.6.1 Privilege Escalation

There are two common types of privilege escalation attacks according to whether the exploitation strategy focuses on inter-process capability leakage or system vulnerabilities. Approaches such as XManDroid [Bugiel et al., 2011a], Woodpecker [Grace et al., 2012a], Elish et al. [Elish et al., 2013] or CHEX [Lu et al., 2012] focus on the first class, while others such as [Checkoway et al., 2010] concentrate on the latter.

XManDroid [Bugiel et al., 2011a] is a privilege escalation detection tool for Android OS devices. Dynamic analysis is used to identify covert channels using DFG. Woodpecker [Grace et al., 2012a] is capable of identifying both explicit and implicit leakage by combining static with dynamic analysis. Static analysis is used to identify possible execution paths by means of CFG, and inter-procedural data flow analysis is used to filter out non-dangerous paths. Additionally, app permissions are examined to broaden leakage search. Similarly, Elish et al. [Elish et al., 2013] use DDG providing user-interaction dependencies of more than 1000 benign and malign apps, while CHEX [Lu et al., 2012] employs system dependence graphs over more than 5000 applications from *Google Play*.

ROPdefender [Davi et al., 2011b] is a generic ROP detection tool for Windows

and Linux–based OS capable of enforcing a return address check. Although ROPdefender is not built for smart devices, the proposed framework can be applied in this context.

2.4.6.2 Grayware

As discussed early in this chapter, grayware poses a serious challenge to privacy leakage detection system. Several approaches have focused on detecting such privacy leakages, such as TaintDroid [Enck et al., 2010] for Android OS devices and PiOS [Egele et al., 2011] for iOS.

TaintDroid [Enck et al., 2010] uses dynamic taint analysis to track sensitive information. It monitors variables, methods, files, and messages throughout the program execution according to data flow rules, and label the variables as they use the sensitive data. When a piece of sensitive information attempts to leave a taint sink, e.g., through the network interface, TaintDroid requests user consent to do so. The authors studied 30 popular applications, showing that at least 20 of them misused users' private information. Experiments also show that TaintDroid incurs 14% CPU and 4.4% memory overhead. A major limitation of TaintDroid is its inability to distinguish between legitimate and non-legitimate exfiltrations, especially when facing grayware. In fact, their experiments show that 37 out of 105 instances (35%) were incorrectly classified as false positives. Additionally, techniques such as tainting can be circumvented through leaks via implicit flows, i.e., using program control flow to disclose information.

AppProfiler [Rosen et al., 2013] uses dynamic tainting analysis along with static analysis to extract privacy-related behaviors. The scheme builds a *knowledge base*

that maps application behaviors with API calls observed during static analysis, providing the user with valuable information about their apps.

Finally, PiOS [Egele et al., 2011] is an information leakage detection system for iOS devices that uses static analysis on apps. PiOS constructs CFG paths from the sources of sensitive information to data sinks by means of data-flow analysis. So far, static analysis of iOS apps does not have to face the obfuscation challenge, as obviously obfuscated apps would not pass the revision process. However, this might change in the coming years if non-walled-garden models such as *Cydia* gain popularity.

2.4.6.3 Battery-depletion

Traditional anomaly and misuse detection techniques have not paid much attention to unknown energy-depletion attacks. In this regard, Kim et al. [Kim et al., 2008] proposes a power-aware malware detection system for smart devices. It uses dynamic analysis to monitor power samples and build a consumption model. Power signatures are generated from monitoring malicious samples in the device, and results are analyzed in the device or in the cloud using noise filtering and data compression algorithms. After building the model, malware is identified by using χ^2-distance and comparing the results with a set of signatures.

3

Maldroid Lab: Research Malware Lab for Smart Malware Analysis and Detection

3.1 Introduction

The analysis of smart malware is currently constrained by the lack of a versatile and multipurpose laboratory for testing new research proposals. In this chapter, we describe the architecture of a framework gathering together the most cutting-edge tools for analyzing and dissecting Android malware.

This Chapter introduces Maldroid Lab, a framework aiming at providing grounds for smart malware research. Maldroid Lab gathers together several monitoring, analysis, and identification systems. On the one hand, it includes a number of open source *static* and *dynamic* tools over a virtual device manager [Android, 2014]. On the other hand, it also extends current systems and implements new functionalities, such as a proof-of-concept of a *cloud clone* system [Chun et al., 2011; Kosta et al.,

2012; Portokalidis et al., 2010].

Maldroid Lab is implemented using generic *Java* and *Python* components and it has been deployed with a sizeable dataset of both legitimate and malicious real-world samples. More precisely, the lab currently compiles over 25K apps from legitimate markets and 25K malicious apps. For the former, Google Play as well as Aptoide are constantly crawled to retrieve new samples. Similarly, we query Android Malware Genome Project[1], Virus Share[2] and Contagio Mobile[3] for the latter. Figure 3.1 presents the architecture of our Maldroid Lab. All this together constitutes a research laboratory for testing new malware analysis techniques and will serve as a building block for the experimentation tasks of each contribution presented in this Thesis.

The architecture of Maldroid Lab has been designed to have the following features:

- Facilitate the tasks of extracting assets, components, and resources from Android apps.
- Automate the process of unpackaging and repackaging Android apps.
- Guarantee the isolation of a fully controlled Android environment for testing apps via virtualization.
- Allow the dynamic allocation of virtual devices and the installation of apps automatically.
- Optimize the execution of such virtual devices using parallelization.

[1] http://www.malgenomeproject.org/
[2] http://virusshare.com/
[3] http://contagiominidump.blogspot.com.es/

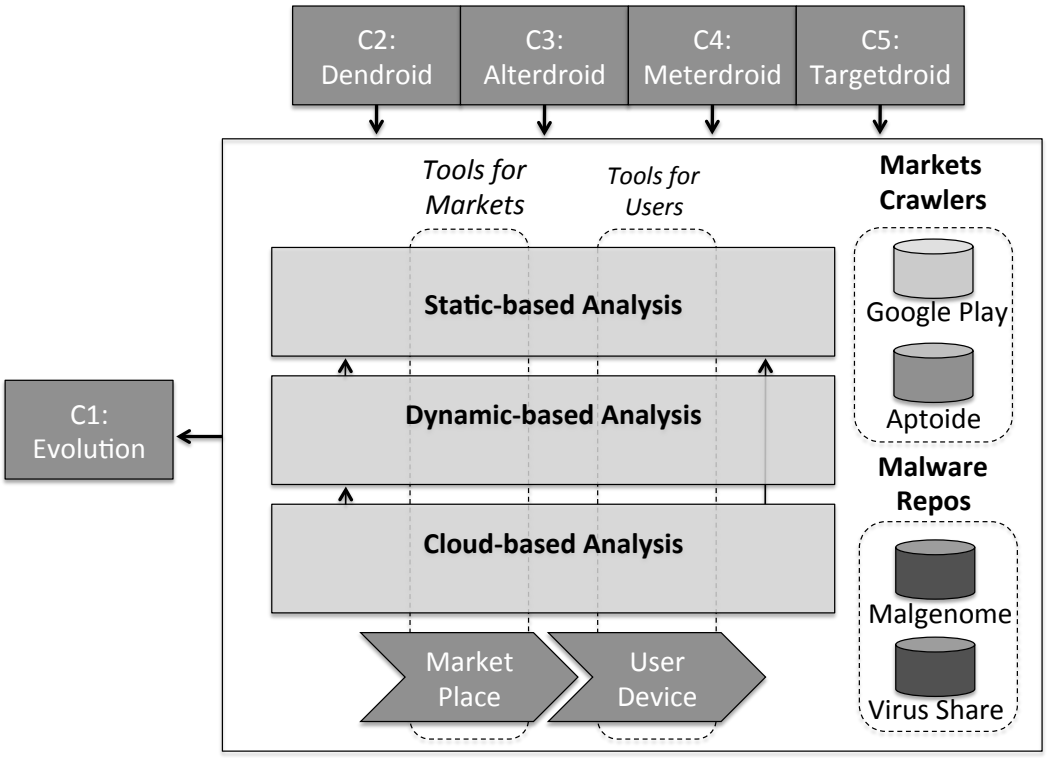


Figure 3.1: Maldroid Lab's architecture in a nutshell.

- Provide the injection of Graphical User Interface (GUI) events, as well as other contextual information such as GPS locations or text SMS messages.

- Allow the execution of certain security tasks over synchronized replicas maintained in the cloud.

The remaining of this chapter is organized as follows. Sections 3.2 and 3.3 describe the main static- and dynamic-analysis tools deployed in this research laboratory, respectively. In Section 3.4 we describe our cloud-based system, which is used for offloading certain tasks from the device to the cloud. Finally, Section 3.5 describes several online repositories used to retrieve both legitimate and malicious apps.

3.2 Static Analysis

Apps are statically analyzed using several techniques aiming at unpackaging and disassembling apps. In our lab, this process is mainly performed using Androguard [Desnos, 2014]. For unpackaging and repackaging apps into a modified app, we use ApkTool [Panxiaobo, 2014] and dex2jar [Alll and Tumbleson, 2014] tools.

Monkey [Android, 2014] and AndroidViewClient [Milano, 2014] are used to generate a common sequence of events to interact with the apps. These events should be generated specifically for each test to intelligently drive the GUI exploration [Rastogi et al., 2013a; Zheng et al., 2012], i.e., to test code implementing different functionalities of the app. In its current implementation, Maldroid Lab uses Monkey and AndroidViewClient to generate five classes of events: *activity launch*, *service launch*, *action buttons*, *screen touch*, and *text input*. We also use Culebra [Milano, 2014] to create AndroidViewClient scripts for further automating the analysis.

We then describe in some detail most popular tools deployed in this lab. Further information about the particularities of each tool can be found in the references given throughout the document.

3.2.1 Androguard

Androguard [Desnos, 2014] is an interactive-oriented static analysis tool for third-party Android applications. It allows to disassemble apps and access their components throughout its API[4]. Androguard's API also provides access to each attribute of the binary code, such as classes, methods, and variables. The main features of its API are:

[4] http://doc.androguard.re/html/index.html

- **APK**. The Android Application Package (APK) is a file format used to distribute Android apps from the markets to the devices. This package is an archive in JAR format containing a number of files and a well-structured directory hierarchy. Examples of files included in an APK file are: the Android Manifest file, the executable classes file, and other precompiled binaries and raw resources. Androguard allows to unpackage all these components and access them through the APK library.

- **DVM**. The Dalvik Virtual Machine (DVM) is a component of Android OS responsible of running the apps on the device. Each Android APK packages a DVM file—known as Dalvik Executable Format (DEX)—containing the compiled Android application code. This component of Androguard disassembles the DEX file and provides access to its components. More precisely, it allows to retrieve Java Annotations (metadata) about a program, the name and size of its classes, methods, and variables, among other static features from the DVM.

- **Analysis**. This library interprets Dalvik's code and provides a semantic analysis of the DVM. It allows to identify where permissions are used in a specific app and when special libraries (such as *crypto* or *reflection* libs) are used. Additionally, it also provides a Control Flow Graph (CFG) representation of the Dalvik code flow. CFGs provided by Androguard are based on a grammar proposed by Cesare and Xiang [Cesare and Xiang, 2010] and shown in Figure 3.2.

- **Bytecode**: The Dalvik code executed by the DVM is a compact and efficient instruction set (numeric codes, constants, and references) that encodes executable programs into a portable language called bytecode. This bytecode is

translated into native machine code at run time. This facilitates the portability of the bytecode itself across different hardware-specific platforms. However, it also makes easier the reverse-engineering analysis of Android apps. This component of Androguard provides a number of methods that aid bytecode analysis.

Grammar:

```
    Procedure  ::=  StatementList
StatementList  ::=  Statement | Statement StatementList
    Statement  ::=  BasicBlock | Return | Goto | If | Field | Package | String
       Return  ::=  'R'
         Goto  ::=  'G'
           If  ::=  'I'
   BasicBlock  ::=  'B'
        Field  ::=  'F'0 | 'F'1
      Package  ::=  'P' PackageNew | 'P' PackageCall
   PackageNew  ::=  '0'
  PackageCall  ::=  '1'
  PackageName  ::=  Epsilon | Id
       String  ::=  'S' Number | 'S' Id
       Number  ::=  \d+
           Id  ::=  [a-zA-Z]\w+
```

Examples:

CC_1 `B[P0P1]B[I]B[P1R]B[P1P1I]B[P0SP1P1P1]B[P1G]|B[F1P1R]`

CC_2
```
B[SSF1F0P1SF0SP1P1I]B[SP1P1F1SP1F1F0I]B[F0P1I]B[F0SP1]B[]
B[P1SP1SP1F1SF0P1I]B[F0I]B[F0P1I]B[F1F0P1P1I]B[F0P1I]B[]B[F0P1]
B[F0I]B[S]B[P1I]B[F0P1]B[I]B[P1F0P1P1F0P1I]B[F0P1P1I]B[F0P1I]
B[]B[F0P1F0P1]B[P0F0P1P1SP1F0P1SP1F0P1SP1F0P1P1F0P1F0P1S]
```

CC_3 `B[P1SF1R]`

Figure 3.2: CFG grammar used by Androguard to extract code structures.

3.2.2 ApkTool

ApkTool [Panxiaobo, 2014] is a reverse engineering tool for third-party Android applications. This tool allows to decode Android apps into Smali code[5]. It also facilitates the modification of the app or the injection of new code before repackaging it.

Smali is a DEX code disassembler that transforms bytecode into a syntax similar to the one used in Jasmin's[6] and dedexer's[7] project. This syntax aims at alleviating the complexity of exploring Java Virtual Machine binaries. Thus, ApkTool allows to reconstruct the original resources into a human-friendly format to facilitate reverse engineering of the code as shown in Figure 3.3. This example shows a code fragment obtained from an Android malware sample known as DroidKungFu.

We then describe the main functions of ApkTool:

- **Decompile**. It performs the inverse operation to that of Dalvik's bytecode compiler and the APK packaging. The resulting folder contains the manifest of the app, all Java classes in Smali language, as well as assembled resources, libraries, and assets.

- **Recompile**. Transforms Smali source code classes resulting from the previous step—together with any other resources contained in the app—in an Android APK file ready to be executed in the device. This new APK may well be different from the original one, e.g., it can contain piggybacked functionality.

[5] https://code.google.com/p/smali/
[6] http://jasmin.sourceforge.net/
[7] http://dedexer.sourceforge.net/

```
.method static constructor <clinit>()V
            new-array v0, v0, [B
            fill-array-data v0, :array_0
            sput-object v0, Lcom/google/ssearch/Utils;->defPassword:[B
            .line 40
            return-void
            .line 228
            :array_0
            .array-data 0x1
                        0x46t
                        0x75t
                        0x63t
                        0x6bt
                        0x5ft
                        0x73t
                        0x45t
                        0x78t
                        0x79t
                        0x2dt
                        0x61t
                        0x4ct
                        0x6ct
                        0x21t
                        0x50t
                        0x77t
            .end array-data
.end method
```

Figure 3.3: Excerpt of an Android malware sample called DroidKungFu extracted with ApkTool and represented in Smali syntax. This piece of code is used by the malware sample to decrypt an exploit distributed together within the APK assets.

3.2.3 Monkey and Monkeyrunner

Monkey and Monkeyrunner [Android, 2014] are two Android Developer tools for automatically testing Android apps. Monkey generates dummy random events to interact with the Operating System. These events typically include GUI actions such as touch, press a button, etc. Monkeyrunner provides the developer with a Python API to interact with the running apps and control the device from the command line. The main components of Monkeyrunner are:

- **Runner**. This component provides a number of utility methods such as communicating with the device, creating user interfaces, and displaying built-in help.

- **Device**. This component facilitates the installation and removal of Android packages. It also provides the appropriate interface for starting Android Activities, sending keyboard or touch events to an app, etc.

- **Image**. This component provides access to the device for capturing screenshots, converting bitmap images to various formats, and comparing two *MonkeyImage* images. This component is very useful for monitoring changes in an Activity running at a given time instant.

3.2.4 AndroViewClient

AndroViewClient is a Python tool that facilitates the creation of scripts for interacting with the device. A remarkable feature of AndroViewClient is its ability to retrieve a tree view of the UI-components displayed on the device at any given moment. For instance, given an Activity, AndroViewClient allows to retrieve which other clickable views are nested into this one. Then, it allows the user to interact with those components by, for instance, clicking them or inserting text into a TextBox.

We instrumented our sandbox with AndroViewClient in order to generate smart UI-interactions. Figure 3.4 depicts a code fragment snipped from our middleware using AndroViewClient. This piece of code provides our lab with the capability to interact with a particular view.

```
    '''
      Dump current window and interact with its children
    '''
      def interactDumpWindow(self, serialno, vc, time):
          windows = vc.list()
          for wld in windows.keys():
              views = vc.dump(window=wld, sleep=time)
              for view in views:
                  if not self.running:
                      break
                  self.interactView(vc, view)
     '''
        Interact with a given View
     '''
     def interactView(self, vc, view, text='InputText'):

          if 'android.widget.EditText' == view.getClass():
              view.type(text)

          if view and view.isClickable() and
            not view.getId() in self.interacted:
              print view.getId(), view.getText()
              view.touch()
              self.interacted.append(view.getUniqueId())
```

Figure 3.4: Code snipped of our middleware using AndroViewClient's API to interact with an app running in the device.

3.3 Dynamic Analysis

We have used an open source dynamic analysis tool called Droidbox [Lantz, 2014] to monitor various activities that can be used to characterize app behavior and tell apart benign from suspicious behavior [Suarez-Tangil et al., 2014b]. We then describe Droidbox together with another tool, called TaintDroid, that is instrumental for detecting information leakage.

3.3.1 Droidbox

Droidbox is a dynamic analysis tool that allows the execution of Android apps and provides a variety of data about how an app is behaving. More precisely, Droidbox monitors the execution of 11 different activities:

- *crypto*: generated when calls to the cryptographic API are invoked.
- *netopen*, *netread*, *netwrite*: associated with network I/O activities (opening a connection, receiving, and sending data).
- *fileopen*, *fileread*, *filewrite*: associated with file system I/O activities (opening, reading, and writing a file).
- *sms*: generated whenever a text message is sent or received.
- *call*: generated whenever a call is made or received from the device.
- *leak*: generated when a leakage of private information has occurred. This is determined using tainting analysis [Enck et al., 2010].
- *dexload*: generated when native code is loaded dynamically.

We have extended Droidbox to allow the extraction of these activities programatically.

3.3.2 Taintdroid

As introduced in Chapter 2, Taintdroid [Enck et al., 2010] uses dynamic taint analysis to track sensitive information throughout a program execution. Taintdroid instruments the DVM interpreter to provide the device with a variable-level tracking

system, as well as message- and file-level tracking. This enhancement offers a valuable awareness of an app's information flow during its execution. Figure 3.5 depicts Taintdroid's architecture as illustrated by Enck et al. in [Enck et al., 2010].

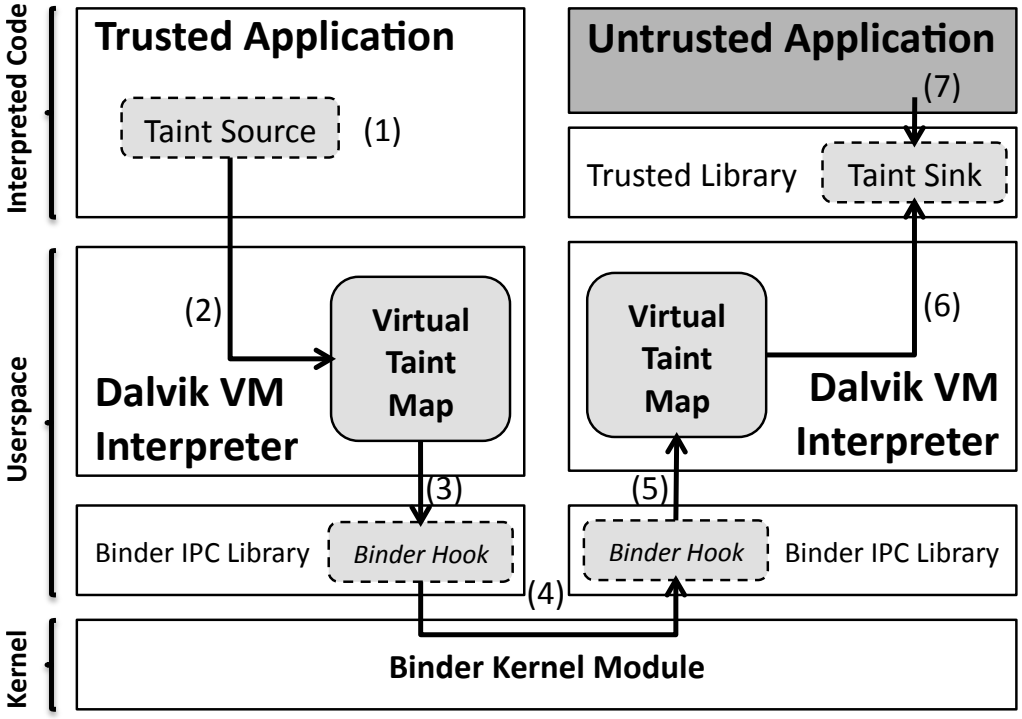


Figure 3.5: Taintdroid's architecture as illustrated in [Enck et al., 2010].

Taintdroid source code is available at the Author's site[8] for several versions of Android such as the ones used by our sandbox, i.e., Android 2.1 and Android 2.3. In our research lab, we use a version of Taintdroid distributed with Droidbox.

3.4 Cloud Analysis and Consumption Metering

Apart from traditional static and dynamic analysis techniques, a number of recent works have opted for a radically different approach based on maintaining a synchro-

[8] http://appanalysis.org/

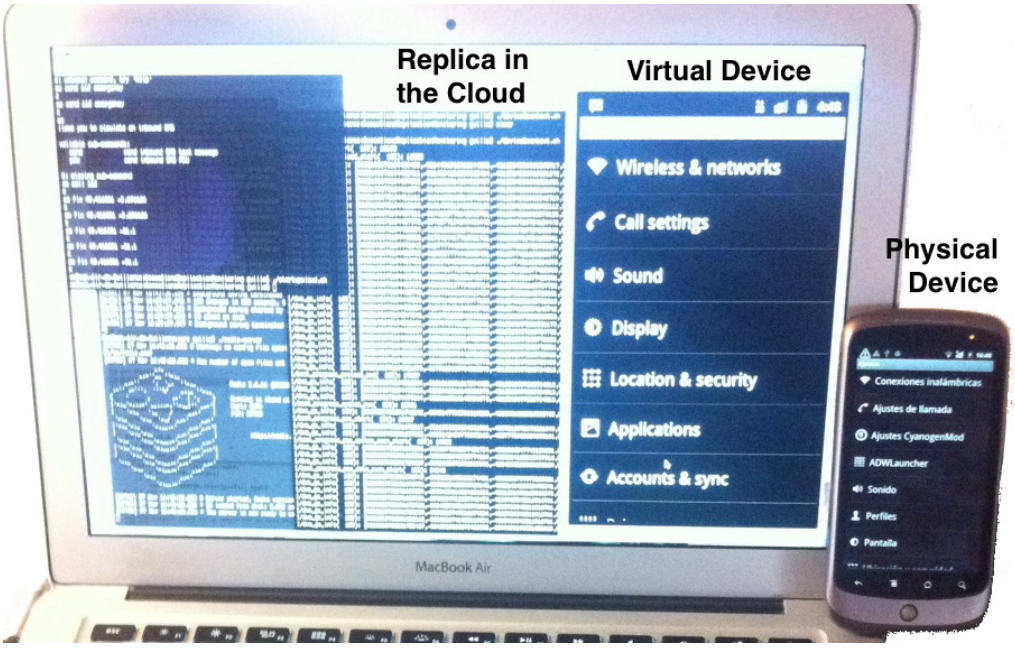


Figure 3.6: Proof of concept of a clone cloud system.

nized replica of the device in the cloud. Paranoid Android [Portokalidis et al., 2010], Secloud [Zonouz et al., 2013] and CloudShield [Barbera et al., 2013] are illustrative examples of such systems. In these cases, all security-related tasks, including monitoring, analysis, and detection can be performed in an environment not exposed to battery or computational constraints. Furthermore, multiple detection techniques can be applied simultaneously, as the clone can be easily replicated. Maldroid Lab implements a proof-of-concept cloud cloning system (see Figure 3.6) based on the aforementioned approaches. We then describe the components of our *Clone Cloud* system:

- **Physical device**. We instrumented a Google Nexus One phone with various monitoring tools that collect user events, the context, etc. and transmit them to the cloud. For this purpose, we used a combination of logcat and getevent tools from Android Debug Bridge (ADB) [Android, 2014].

- **Cloud virtual device**. In the cloud-end, a middleware implemented in Python processes all inputs received, generates the associated models, and runs the simulation. We inject events and contexts from the physical device into apps using both Monkeyrunner [Android, 2014] and the Android emulator console [Android, 2014].

One critical issue with these approaches is that keeping the clone synchronized involves a constant exchange of activity update packets. For example, experiments on Paranoid Android show that synchronizing the device with the cloud replicas require exchanging traces at 2 KB/s for high-load scenarios and at 64 B/s for idle operation. This definitely consumes power, although it may be worth doing if the clone is a subject to intensive monitoring.

We also instrumented our physical device with a tool for estimating the energy consumption of the device. The technical issues on metering and modeling power consumption in mobile devices have received much attention lately. Built-in meters in platforms such as Android provide a coarse power profile and are inadequate for most applications. Our choice of Appscope [Yoon et al., 2012] in this Thesis is motivated by its accuracy, and also because it provides separate energy consumption for each app and process, detailing how much corresponds to CPU usage, networking, touchscreen, etc. Other alternatives include PowerTutor [Zhang et al., 2010], Systemtap [Dediu, 2014a], Eprof [Pathak et al., 2012], and also the schemes discussed in [Dong and Zhong, 2011; Hao et al., 2012; Nagata et al., 2012; Pathak et al., 2011].

We then describe Appscope and present a middleware called Crowdcosec implemented in this Thesis for offloading on-board information to the cloud.

3.4.1 Appscope

AppScope [Yoon et al., 2012] is an energy metering framework that monitors the kernel activity of Android devices. AppScope collects usage information from the monitored device and estimates the consumption of each running application using an energy model given by DevScope [Jung et al., 2012]. AppScope provides the amount of energy consumed by an app in the form of several time series, each one associated with a component of the device (CPU, Wi-Fi, cellular, touchscreen, etc). AppScope uses event-driven monitoring method that produces low overhead and provides high accuracy. In fact, its authors report that AppScope incurs approximately 35mW and 2.1% in power consumption and CPU utilization overhead, respectively.

We have instrumented our Google Nexus One phone with AppScope. Figure 3.7 depicts the information provided by AppScope when measuring the power consumption of an app while being executed in the device. AppScope provides information about the power consumed by different applications running in the device. Additionally, it also offers information about the energy consumed by each individual process executed by every app. We have also implemented a number of shell scripts to automatically collect details of the energy consumed by the device from AppScope's logs. (Incidentally, this process turned out to be very challenging due to the lack of documentation describing the format of AppScope's logs.)

3.4.2 Crowdcosec

We have tested several open source sensing frameworks such as Funf[9] from MIT, SystemSens [Falaki et al., 2011] and ProfileDroid [Wei et al., 2012]. Our experience

[9] http://www.funf.org/

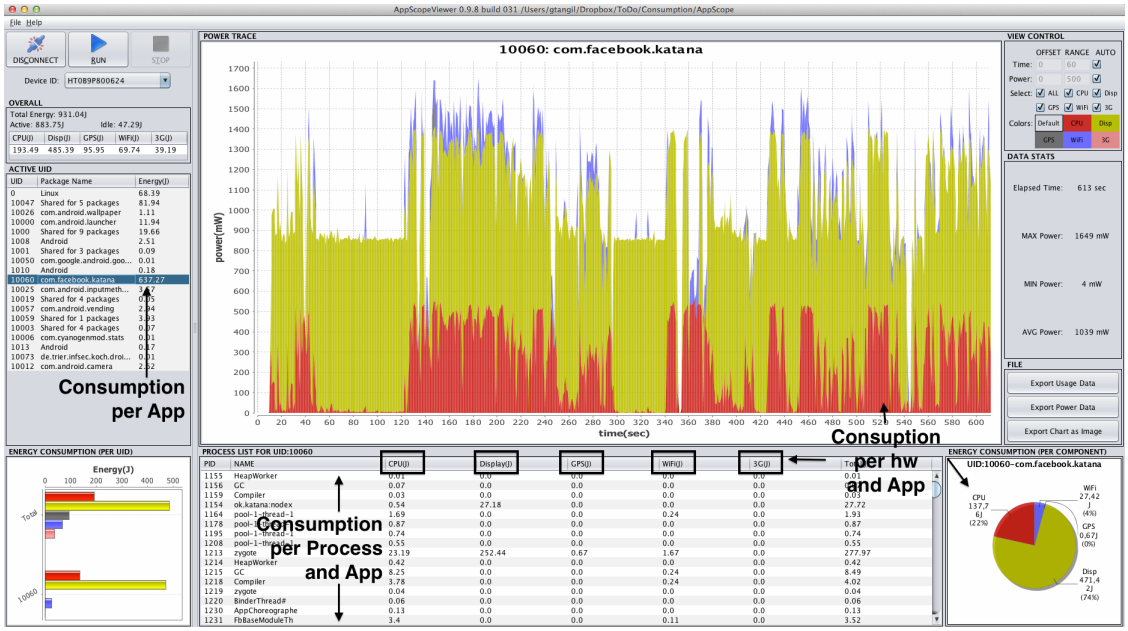


Figure 3.7: Metering Facebook's power consumption with Appscope.

was that all these frameworks were still at their first stages by the time we tested them. Thus, we implemented our own sensing framework to retrieve system information from a crowd of devices. In particular, we are currently extracting system calls (*syscalls*) generated by the apps. We use for this purpose a tool called strace[10]. However, we can easily extend our framework to retrieve other information from the devices.

Our *syscall* module, only runs in *rooted* devices with super user privileges. Collected *syscalls* are processed and sent to a remote server implemented over Apache[11]. Crowdcosec allows also to process collected traces locally in the device. In this regard, we instrumented our Crowdcosec app with a stripped version of Weka [Hall et al., 2009] capable of running any Machine Learning algorithm implemented in Weka.

[10] strace is a debugging tool for Linux and some other Unix-like systems that allows to monitor the system calls used by a program. strace for Android is available online at `http://benno.id.au/blog/2007/11/18/android-runtime-strace`.

[11] `http://www.apache.org/`

3.5 Onlinet Markets and Malware Repositories

As introduced in Chapter 2, Android's architecture implements an *open-market* model. Therefore, users are free to download applications from any market. We then describe the legitimate and malicious repositories used to carry out experimentation in our laboratory. We also introduce an open source remote access malware called Androrat used in this Thesis.

3.5.1 Crawling Online Markets

There are a substantial number of Android application markets with a variety of apps available for all users. Typically, legitimate developers upload paid apps to Android's official market, i.e., Google Play[12]. Contrarily, unofficial markets such as Aptoide[13] generally host the "very same" applications for free. To retrieve a large amount of samples, we have crawled the following two markets and downloaded apps from both of them:

- **Google Play**. Google Play is the official Android distributor for Android apps. Users are able to search for apps and download them, as well as get access to lists of apps ranked by popularity.

  We have implemented a Google Play crawler in Python to automatically download apps from this market. The crawler uses an unofficial open-source API[14] for automatically querying Google Play.

[12] http://play.google.com/store/apps
[13] http://www.aptoide.com/
[14] https://code.google.com/p/android-market-api/

- **Aptoide**. Aptoide is a popular framework for deploying your own alternative market. It has so far over 110K stores and a total of almost 200K apps and about 800M downloads.

  We have implemented an Aptoide crawler in Ruby to automatically download apps from this market. The crawler uses the official Aptoide APIs to obtain metadata from the markets and get the location of the app. Then, we use our crawler to retrieve new apps.

We have currently crawled about 25K apps from both markets. We mainly have apps from Aptoide, as Google Play limits the number of downloads per day and user.

3.5.2 Malware Repositories

The growth of Android malware has come hand in hand with the proliferation of online repositories sharing the latest specimens. There are a number of public and/or private malware repositories such as:

- **Malware Genome Project**. The *Android Malware Genome Project*[15] is a malware repository that covers the majority of malware families for Android OS. All these samples have been collected and characterized into families by Zhou and Jian in [Zhou and Jiang, 2012]. We accessed this repository at the end of 2011. Back then it contained 1247 malicious apps grouped into 49 different families. These samples included specimens with a variety of infection techniques (repackaging, update attacks, and drive-by-download) and payload functionalities (privilege escalation, remote control, financial charge, and private information exfiltration).

[15] Available at `http://www.malgenomeproject.org`

- **Virus Share**. *Virus Share*[16] is a repository of malware samples for security researchers that counts with a massive number of malware samples —over 15M of them. We visit this repository on a regular basis to retrieve the latest samples found in the wild.

- **Contagio Mobile**. *Contagio Mobile*[17] is a public malware repository managed by a group of independent security researchers and with great amount of support within the malware community. We also visit this repository regularly.

In total, our malware repositories in the lab currently have about 25K malware samples.

3.5.3 Open Source Malware Remote Access Tool

Androrat [Bertrand et al., 2014] is an an open source malware Android Remote Access Tool (RAT). RAT tools provide a backdoor to a remote operator, enabling access to the device and its personal data. Androrat is provided with two different components: (i) the remote manager running on a server, and (ii) the local agent running on the device. We then describe each of these two components:

- **Server**: This component implements command and control and allows the user to remotely control numerous devices. The devices are listed dynamically as new users are connected.

- **Client**: This component implements all functionality required to provide the server with the information requested. More precisely, Androrats counts in its current implementation with the following capabilities:

[16] http://virusshare.com/
[17] http://contagiominidump.blogspot.com.es/

- Get contacts (and all their information).
- Get call and messages logs.
- Get geolocation by GPS/Network.
- Monitor received messages and phone state in real time.
- Take a picture from the camera and stream sound or video.
- Do a toast.
- Send a text message or make a call.
- Open an URL in the default browser.

Figure 3.8 shows a snapshot of this tool running over our lab.

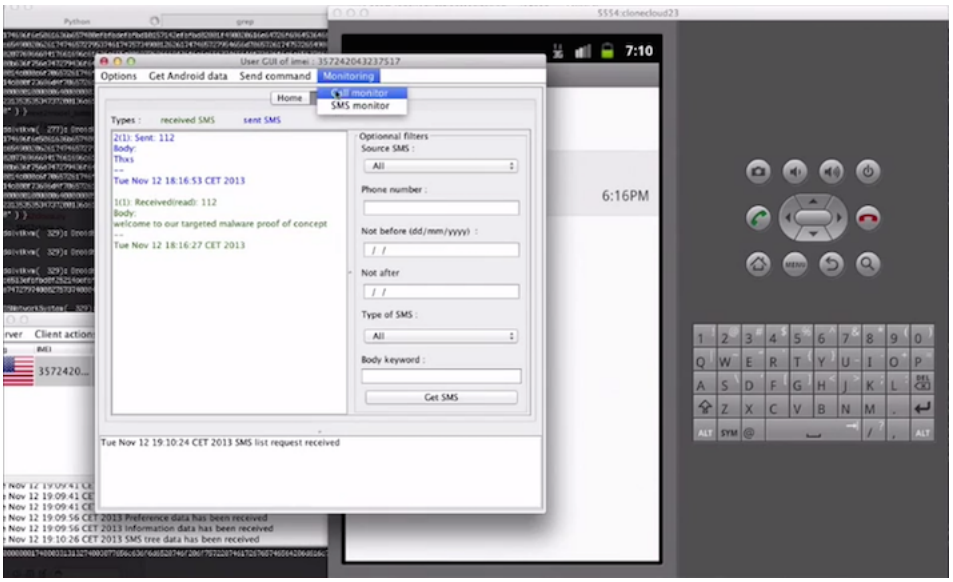


Figure 3.8: Exfiltrating personal information (SMSs) with Androrat.

II

Static-based Analysis

4

A Text Mining Approach to Analyzing and Classifying Code Structures in Malware Families

4.1 Introduction

The impressive growth both in malware and benign apps is making increasingly unaffordable any human-driven analysis of potentially dangerous apps. For instance, when confronted with a continuously growing stream of incoming malware samples, it would be extremely helpful to differentiate between those that are minor variants of a known specimen and those that correspond to novel, previously unseen samples. Grouping samples into families, establishing the relationships among them, and studying the evolution of the various known "species" is also a much sought after application.

Problems similar to these ones have been successfully attacked with Artificial Intelligence and Data Mining techniques in many application domains, including mal-

ware detection [Egele et al., 2012]. For instance, machine learning [Hou et al., 2010], data mining [Liao et al., 2012; Thiruvadi and Patel, 2011], expert systems [Sahin et al., 2012], and clustering [Delany et al., 2012], have been proposed to assist the analyst in classifying the malware. We refer the reader to Chapter 2 for an overview on automated malware analysis techniques.

In this chapter, we explore the use of text mining approaches to automatically analyze smartphone malware samples and families based on the code structures present in their software components. Such code structures are representations of the Control Flow Graph (CFG) of each method found in the app classes [Cesare and Xiang, 2010; Grace et al., 2012a]. A high level overview of the main building blocks and salient applications of our approach, namely Dendroid [Suarez-Tangil et al., 2014c], is provided in Figure 4.1. During the modeling phase, all different code structures are extracted from a dataset of provided malware samples. A vector space model is then used to associate a unique feature vector with each malware sample and family. This vector representation is then used to illustrate two main applications:

- Automatic classification of unknown malware samples into candidate families based on the similarity of their respective code structures. Our classification scheme involves a preparatory stage where the sample is transformed into a query in the text mining sense. Thus, a slight variation of this process can be used to search for a set of given code structures in a database of known specimens, a task that could be remarkably useful for malware analysts and app market operators.

- We show how it is possible to perform an evolutionary analysis of malware families based on the dendrograms obtained after hierarchical clustering. The

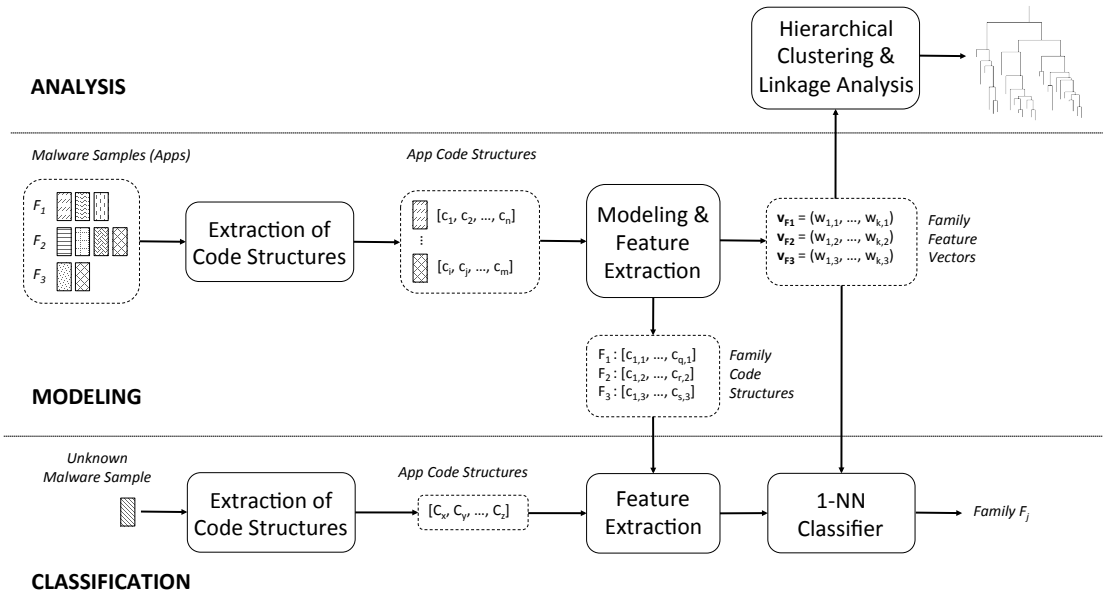


Figure 4.1: Overview of Dendroid's architecture.

process is almost equivalent to the analysis of the so-called phylogenetic trees for biological species [Ruzgar and Erciyes, 2012], although using software code structures rather than physical and/or genetic features. This enables us to conjecture about evolutionary relationships among the various malware families, including the identification of common ancestors. Additionally, it also enables us to study the diversification process that they may have gone through as a consequence of code reuse and malware re-engineering techniques.

In other domains, many works have applied text mining and information retrieval techniques for decision making and classification, such as for example [Chibelushi et al., 2004], and [Gadia and Rosen, 2008]. Furthermore, recent approaches have also used text mining for detecting similarities [Oberreuter and Velásquez, 2013; Rodriguez-Gonzalez et al., 2013].

Dendroid is novel in two separate ways. On the one hand, to the best of our

knowledge using code structures to characterize Android OS malware families has not been explored before. One major advantage of focusing on the internal structure of code units (methods) rather than on their specific sequence of instructions is an improved resistance against obfuscation (i.e., deliberate modifications of the code aimed at evading pattern-based recognition [Rastogi et al., 2013b]). Furthermore, such structures prove to be particularly useful for the case of smartphone malware, where rapid development methodologies heavily based on code reuse are prevalent. On the other hand, the idea of using text mining techniques to automate tasks such as classifying specimens, searching for code components, or studying evolutionary relationships of malware families is, to our knowledge, novel too. Besides, text mining techniques were developed to efficiently deal with massive amounts of data, a feature which turns out to be very convenient for the problems that we address here.

The remaining of this chapter is organized as follows. In Section 4.2 we describe the dataset of Android malware families used in this work, together with the tools and methodology followed to extract code structures from each app. In Section 4.3 we analyze and discuss various statistical features of the code structures found in the malware instances. Based on our findings from this analysis, in Section 4.4 we propose Dendroid [Suarez-Tangil et al., 2014c], a text mining approach to classify and analyze malware families according to the code structures present in their apps. We first introduce a suitable vector space model, and report experimental results related to classifying instances into families, measuring similarity among families, and using dendrograms to analyze the evolutionary relationships among families. Finally, Section 4.5 concludes the chapter and discusses our main contributions.

4.2 Dataset and Experimental Setting

The work presented in this chapter is largely based on a sizable dataset of real-world Android OS malware samples called *Android Malware Genome Project* and described in Chapter 3. As commented before, this dataset contains 1247 malicious apps grouped into 49 different families. For the purposes of this work, we discarded 16 out of the 49 families as they only contain one specimen each, resulting in a final dataset of 1231 malware samples grouped into 33 families. More details on this will be later provided in Section 4.3.

4.2.1 Extracting Code Structures

One key aspect of our work is the decomposition of an app into a number of constituent code elements referred to as *code chunks*. Each code chunk corresponds to a method associated with a class within the app. Thus, an app will be fragmented into as many code chunks as methods contained in it. Rather than focusing on the specific sequence of instructions contained in a code chunk, we extract a high-level representation of the associated Control Flow Graph (CFG). CFGs use graphs as a representation of the paths that a program might traverse during its execution. Each node in a CFG represents a "basic block", i.e., a piece of code that will be sequentially executed without any jumps. The CFG of a piece of code is explicit in the source code, is relatively easy to extract, and has been extensively used in static analysis techniques [Nielson et al., 1999].

Each malware instance contained in the dataset described above has been first disassembled into Dalvik instructions. We then used Androguard [Desnos, 2014] to extract the code chunks of all malicious apps and compute their structure as

described in Chapter 3. The sequence of instructions contained in a code chunk is thus replaced by a list of statements defining its control flow, such as a block of consecutive instructions (B), a bifurcation determined by an "if" condition (I), an unconditional go-to jump (G), and so on. After parsing each code chunk with this grammar, the resulting structure is a sequence of symbols of varying length (see Figure 3.2 at Chapter 3).

After this process, each malware sample *app* is represented by a sequence:

$$app = \langle c_1, c_2, \ldots, c_{|app|} \rangle, \tag{4.1}$$

where c_i is a string describing the code structure of the i-th method in *app*, and $|app|$ is the total number of methods contained in *app*. In the remaining of this chapter, we will refer to c_i's indistinctly as code chunks or code structures. The resulting dataset of code chunks, grouped by app and family as in the original Android Malware Genome Project, has been made publicly available[1].

4.3 Analysis of Code Structures in Android Malware Families

In this section, we analyze and discuss various statistical features of the code structures found in the malware apps and families of the dataset described above. Our findings will subsequently motivate the use of text-mining techniques for tasks such as, for example, the classification of new apps into candidate malware families or the analysis of similarities among families.

[1] http://www.seg.inf.uc3m.es/~guillermo-suarez-tangil/dendroid/codechunks.zip

4.3.1 Definitions

We are interested in exploring questions such as how large, in terms of number of code chunks (CCs), apps are; what the distribution of CCs across apps and families is; or how discriminant a subset of CCs is for a given family. We next introduce a number of measures that will be later used to perform this analysis.

Definition 1 (CC). *We denote by* $\texttt{CC}(app)$ *the set of all different CCs found in app app. We emphasize that* $\texttt{CC}(app)$ *is a set and, therefore, it does not contain repeated elements.*

Definition 2 (Redundancy). *The redundancy,* $RD(app)$*, of an app app is given by:*

$$RD(app) = 1 - \frac{|\texttt{CC}(app)|}{|app|}, \tag{4.2}$$

where $|app|$ *is the total number of CCs (possibly with repetitions) in app.*

Note that redundancy measures the fraction of repeated CCs present in an app, with low values indicating that CCs do not generally appear multiple times in the app, and vice versa.

Definition 3 (FCC). *The set of family CCs for a family* $\mathcal{F}_i$ *is given by:*

$$\texttt{FCC}(\mathcal{F}_i) = \bigcup_{app \in \mathcal{F}_i} \texttt{CC}(app). \tag{4.3}$$

Definition 4 (CCC). *The set of common CCs for a family* $\mathcal{F}_i$ *is given by:*

$$\texttt{CCC}(\mathcal{F}_i) = \bigcap_{app \in \mathcal{F}_i} \texttt{CC}(app). \tag{4.4}$$

In short, the set $\texttt{CCC}(\mathcal{F}_i)$ contains those CCs found in all apps of $\mathcal{F}_i$. Even though this can be certainly seen as a distinctive feature of family $\mathcal{F}_i$, it does not imply that all those CCs are unique to $\mathcal{F}_i$. For instance, code reuse—which is a recurrent feature of malware in general and, particularly, of smartphone malware—will make the same CCs appear in multiple families.

Definition 5 (FDCC)**.** *Given a set of malware families $\mathcal{M} = \{\mathcal{F}_1, \ldots, \mathcal{F}_m\}$, a set $C = \{c_1, \ldots, c_n\}$ of CCs is fully discriminant for $\mathcal{F}_i$ with respect to $\mathcal{M}$ iff:*

(i) $C \subseteq \texttt{CCC}(\mathcal{F}_i)$*, and*

(ii) $\forall \mathcal{F}_k \in \mathcal{M}, \mathcal{F}_k \neq \mathcal{F}_i : C \cap \texttt{FCC}(\mathcal{F}_k) = \emptyset$.

We denote by $\texttt{FDCC}(\mathcal{F}_i|\mathcal{M})$ *the maximal set of fully discriminant CCs for $\mathcal{F}_i$ with respect to $\mathcal{M}$; that is, $C = \texttt{FDCC}(\mathcal{F}_i|\mathcal{M})$ iff C is fully discriminant for $\mathcal{F}_i$ with respect to $\mathcal{M}$, and for all C' such that C' is fully discriminant for $\mathcal{F}_i$ with respect to $\mathcal{M}$, $C' \subseteq C$.*

Put simply, a set of CCs is fully discriminant for a family $\mathcal{F}_i$ if and only if every CC in the set appears in every app of $\mathcal{F}_i$ and, furthermore, no CC in the set appear in any app of any other family. Consequently, such a set unequivocally identifies the family, provided that it is not the empty set.

4.3.2 Results and Discussion

We computed the various measures and sets described above over all the apps and families in our dataset. Table 4.1 and Figure 4.2 summarize the most relevant results.

The entire dataset contains 84854 different CCs. In terms of number of unique CCs, apps do not display a uniform behavior, neither within the same family nor across families. Apps in some malware families have, on average, only a few different CCs: see for example FakePlayer (6), GPSSMSSpy (13), or SndApps (28). In contrast, others are quite large, such as for example zHash (1348), Pjapps (1160), or DroidKungFu4 (936).

The variance, both of apps' length and redundancy within each family, is generally large, as illustrated by the boxplots shown in Figures 4.2(a) and 4.2(b). This can be explained by a number of factors, including the fact that in many cases malware belonging to the same family appears in very different apps, each one with its own set and distribution of CCs. In general, however, all apps display a redundancy between 0.4 and 0.7 regardless of their size.

The size of the FCC, CCC, and FDCC sets for each family reveal some remarkable details. The number of family CCs (FCC) varies quite significantly across families. Furthermore, such variability seems uncorrelated with the average number of CCs in the apps. The most likely explanation for this has to do with the proliferation and prevalence of each malware family. Families such as AnserverBot, Geinimi, Pjapps, and DroidKungFu appeared in a variety of very popular repackaged apps, and infected a significant number of devices. Thus, finding the same malware in very different apps induces a sharp increase in the size of FCC.

The CCC set removes this diversity and identifies code structures common to all available apps within a family. The size of this set varies across families, being quite low in families where the malware code has undergone significant evolution, possibly after being included in different apps. For example, only 6 CCs appear in each of

		App stats		Family stats		
Family $\mathcal{F}_i$	$\lvert\mathcal{F}_i\rvert$	$Avg\{\lvert\texttt{CC}(app)\rvert\}$	$Avg\{RD(app)\}$	$\lvert\texttt{FCC}(\mathcal{F}_i)\rvert$	$\lvert\texttt{CCC}(\mathcal{F}_i)\rvert$	$\lvert\texttt{FDCC}(\mathcal{F}_i\lvert\mathcal{M})\rvert$
ADRD	22	416	0.59	2726	21	8
AnserverBot	187	367	0.64	17635	44	9
Asroot	8	78	0.57	462	1	0
BaseBridge	122	433	0.53	9918	5	0
BeanBot	8	746	0.68	3081	61	34
Bgserv	9	384	0.53	487	67	34
CruseWin	2	82	0.53	82	82	40
DroidDream	16	302	0.51	2545	10	0
DroidDreamLight	46	529	0.54	3339	40	13
DroidKungFu1	34	501	0.58	7609	10	0
DroidKungFu2	30	295	0.51	2418	9	0
DroidKungFu3	309	872	0.58	19092	48	11
DroidKungFu4	96	936	0.56	9239	19	2
DroidKungFuSapp	3	351	0.66	411	310	0
FakePlayer	6	6	0.73	7	10	2
GPSSMSSpy	6	13	0.44	23	9	3
Geinimi	69	430	0.58	12141	77	37
GingerMaster	4	223	0.64	297	159	108
GoldDream	47	513	0.54	9129	13	3
Gone60	9	35	0.41	56	26	5
HippoSMS	4	148	0.67	262	8	1
KMin	52	502	0.50	795	120	42
NickySpy	2	65	0.71	84	47	34
Pjapps	45	1160	0.58	15128	6	0
Plankton	11	133	0.52	876	14	2
RogueLemon	2	962	0.54	1441	483	321
RogueSPPush	9	365	0.60	633	114	60
SndApps	10	28	0.55	54	20	11
Tapsnake	2	33	0.57	55	12	2
YZHC	22	316	0.48	1704	33	11
Zsone	12	365	0.40	535	338	1
jSMSHider	16	113	0.46	266	64	52
zHash	11	1348	0.56	2344	645	390

Table 4.1: Statistical indicators obtained for all apps and families in the dataset.

the 45 samples of `Pjapps`. On the contrary, apps in unpopular or rare families share essentially the same version of the malware: see for example `zHash`, where all its 11 apps share 645 CCs.

Finally, the rightmost column in Table 4.1 shows the number of fully discriminant CCs for each family. Surprisingly, The `FDCC` set is non-empty for 26 out of the 33

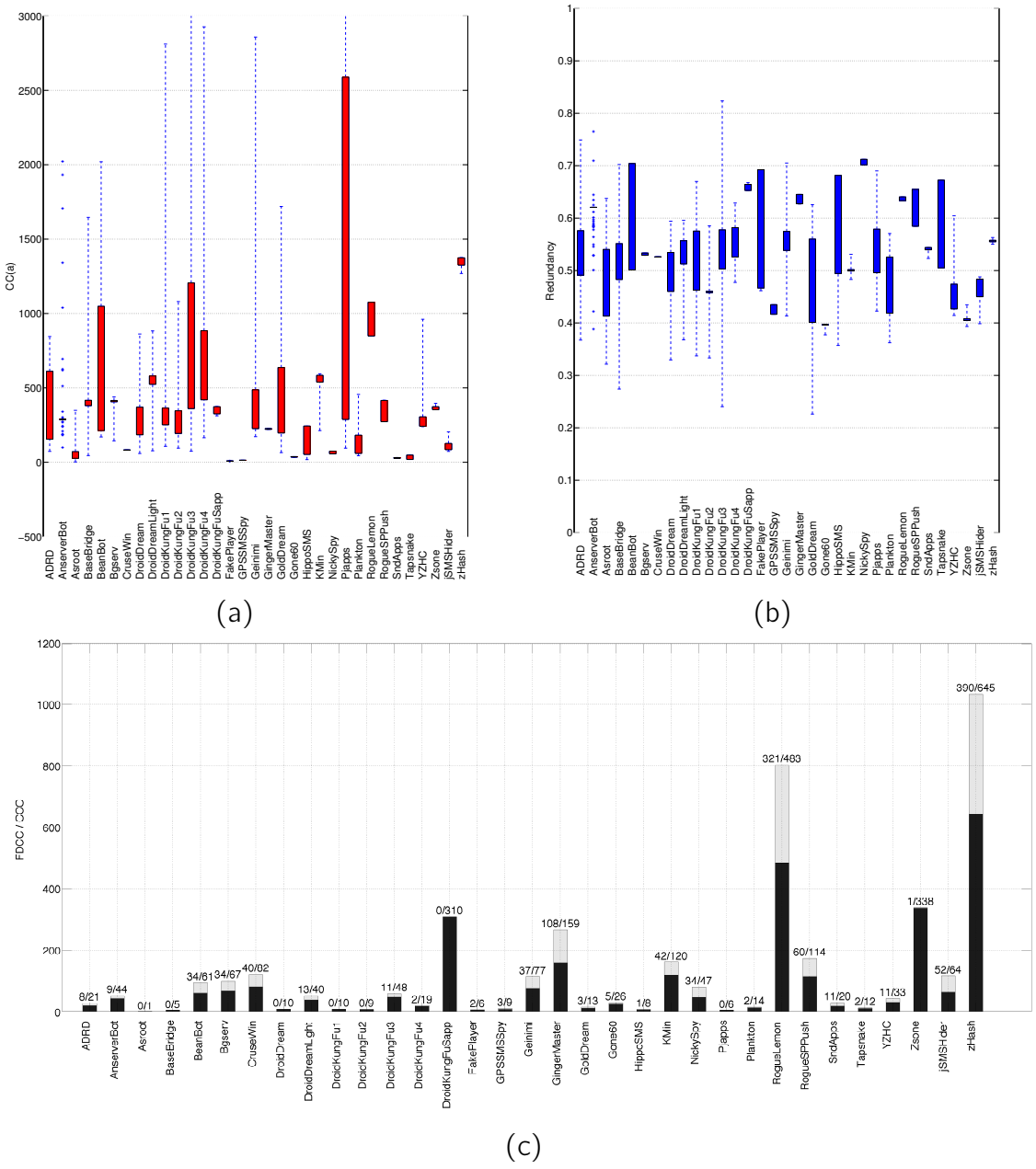


Figure 4.2: Distribution of (a) unique CCs (CC); (b) redundancy (*RD*); and (c) common and fully discriminant CCs for each family (CCC/FDCC).

families. This suggest that, in principle, those CCs might be used as a "signature" to perfectly classify an app into one of those families. We believe, however, that such a scheme would be extremely weak for a number of reasons. One of the most important

shortcomings of using FDCC as the basis to represent malware family features is that it is very fragile: the addition of a new app to a family such that it does not share any CCs with those already in the family automatically makes the CCC set empty, which in turn makes FDCC empty too. Such an app might have actually been incorrectly labeled as belonging to the family, or perhaps carefully constructed to avoid sharing CCs with all other apps. In either case, the characterization of the family would not be useful anymore.

We next study the distribution of CCs across families, which will motivate a more robust representation of family features.

4.3.3 Distribution of Code Structures

Figure 4.3 shows the distribution of CCs as a function of the number of families where they appear. This plot is obtained by iterating over all different code structures and computing, for each one of them, the number of different families where they appear. (A CC appear in a family if it appears in at least one app of that family.) The results reveal that 78.9% of all code structures appear in just one family. Note that this does not mean that such a family is the same, as different code structures may appear in different families. Rather, this value indicates that if a code structure is found in one family, it is unlikely to find that same code structure in an app belonging to a different family. Similarly, the number of code structures that appear in 2, 3, 4, and 5 different families drops to 12.6%, 3.5%, 1.5%, and 1.1%, respectively. Consequently, less than 1% of all available code structures appear in 6 or more different families.

This distribution of code structures across malware families suggests that each family can be sufficiently well characterized by just a few code structures, possibly

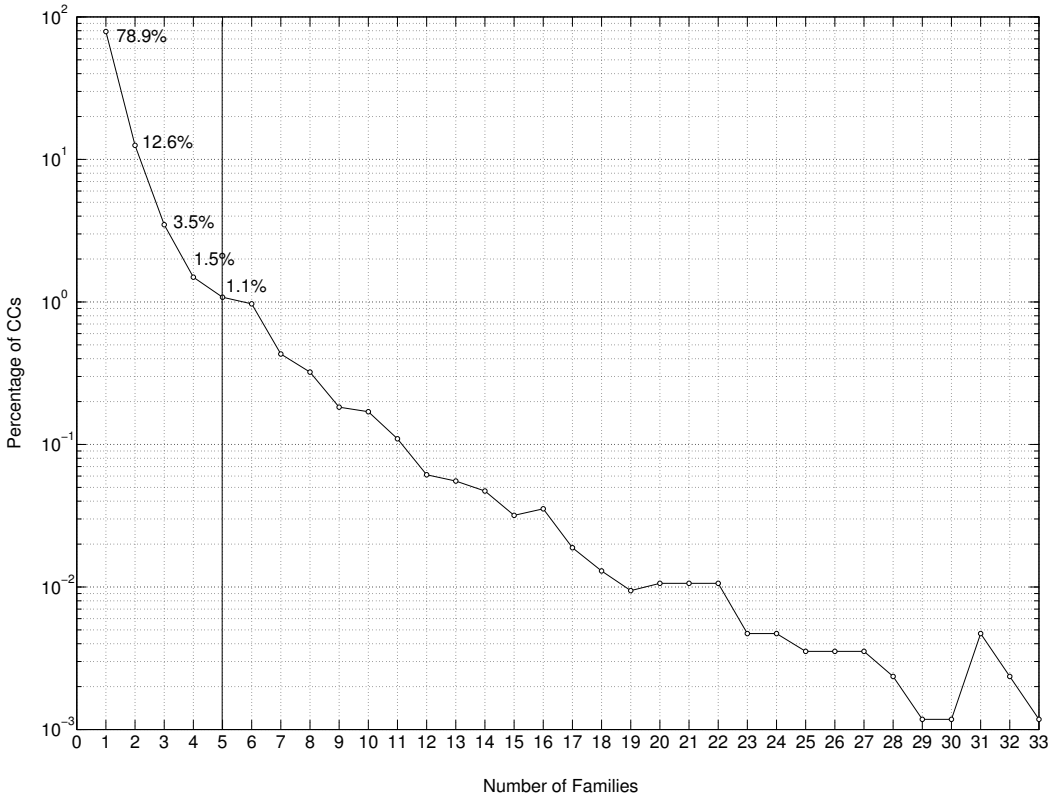


Figure 4.3: Distribution of CCs as a function of the number of families where they appear.

accompanied by some extra information such as the frequency of that code structure in each app of the family, the fraction of apps where it appears, etc. We next elaborate on this.

4.4 Mining Code Chunks in Malware Families

Based on the findings discussed in the previous section, we next describe Dendroid, our approach to analyzing malware samples and families based on mining code structures. We first present the vector space model used and describe the main features of our prototype implementation. Subsequently we present two main applications—

classifying unknown malware apps, and analyzing possible evolutionary paths of malware families—and discuss the experimental results obtained.

4.4.1 Vector Space Model

In this section, we adapt to our problem various numerical indicators well researched in the field of information retrieval and text mining. One central concept in those fields is the so-called Vector Space Model (VSM) [Salton et al., 1975], sometimes known as Term Vector Model, where each object d_j of a corpus is represented as a vector of identifiers

$$\mathbf{d_j} = (w_{1,j}, \ldots, w_{k,j}). \tag{4.5}$$

Each identifier $w_{i,j}$ is a measure of the relevance that the i-th term, m_i, has in object d_j. In the most common setting, objects and terms are documents and words, respectively. Thus, $w_{i,j}$ is an indicator of the importance of word m_i in document d_j.

Many interesting problems related to information retrieval and text mining can be easily reformulated in the VSM in terms of vector operations. For example, the cosine of the angle between two vectors is a good measure of the similarity between the associated documents. Such vector operations are the basis for a number of interesting primitives, such as comparing two documents or ranking various documents according to their similarity to a given query (after appropriately representing queries as vectors too).

One popular statistical indicator used in the VSM is the term frequency-inverse document frequency (tf-idf). Using the notation introduced above, the tf-idf $w_{i,j}$ of term m_i in d_j is the product of two statistics: (1) the term frequency (tf), which

measures the number of times m_i appears in d_j; and (2) the inverse document frequency (idf), which measures whether m_i is common or rare across all documents in the corpus. Thus, a high tf-idf value means not only that the corresponding term appears quite often in a document, but also that it is not frequent in other documents. As a result, one important effect is that the tf-idf tends to filter out terms that are common across documents.

Our proposal essentially mimics the model discussed above. Each family $\mathcal{F}_j$ is represented by a vector $\mathbf{v_j} = (I_{1,j}, \ldots, I_{k,j})$, where $I_{i,j} = \mathtt{I}(c_i, \mathcal{F}_j, \mathcal{M})$ is computed as

$$\mathtt{I}(c_i, \mathcal{F}_j, \mathcal{M}) = \mathtt{ccf}(c_i, \mathcal{F}_j) \cdot \mathtt{iff}(c_i, \mathcal{M}). \tag{4.6}$$

The indicators $\mathtt{ccf}(c, \mathcal{F}_j)$ and $\mathtt{iff}(c, \mathcal{M})$ are approximately equivalent to the tf and idf statistics, respectively, and can be computed as follows.

Definition 6 (CCF)**.** *The frequency of a CC c in a family $\mathcal{F}_j$ is given by*

$$\mathtt{ccf}(c, \mathcal{F}_j) = \frac{\sum_{app \in \mathcal{F}_j} \mathtt{freq}(c, app)}{\max\{\mathtt{freq}(c, app) \ : \ app \in \mathcal{F}_j\}}, \tag{4.7}$$

where $\mathtt{freq}(c, app)$ is the number of occurrences of CC c in app app.

Definition 7 (IFF)**.** *The inverse family frequency of a CC c with respect to a set of malware families $\mathcal{M} = \{\mathcal{F}_1, \ldots, \mathcal{F}_m\}$ is given by*

$$\mathtt{iff}(c, \mathcal{M}) = \log \frac{|\mathcal{M}|}{1 + |\{\mathcal{F}_i \in \mathcal{M} \ : \ c \in \mathtt{FCC}(\mathcal{F}_j)\}|}. \tag{4.8}$$

4.4.2 An example

We next illustrate the model presented above with a numerical example and discuss some relevant features. Assume two different datasets, $\mathcal{M}_1$ and $\mathcal{M}_2$, of malicious apps, with $|\mathcal{M}_1| = 4$ and $|\mathcal{M}_2| = 400$. Given a CC c_i, we can easily see how each family feature vector varies according to the relevance of c_i.

On the one hand, when c_i is a rather common CC (see Figure 4.4a), i.e., it appears in most families, the `iff` value quickly vanishes (see Fig. 4.4b). Similarly, it can also be observed how the components of a family vector grow when the frequency of a CC increases, as shown in 4.4a. On the other hand, when c_i is a very uncommon CC, the `iff` value grows significantly: see, e.g., Figure 4.4 where $\texttt{iff}(c_i, M_2)$ is 16 times larger than $\texttt{iff}(c_i, M_1)$. The overall result is that the relevance of a CC is strongly influenced by its frequency across families. Thus, CCs that are common to many families have a low influence in the family feature vector, even if they are very frequent.

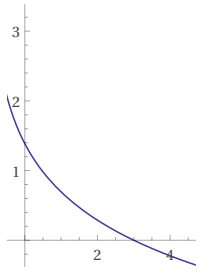
4.4.3 Implementation

We have built a Java implementation of the VSM discussed above and applied it over all families in our dataset to obtain a family feature vector for each of them. The process is described by the algorithm shown in Figure 4.5 and outputs one vector $\mathbf{v_j}$ for each malware family $\mathcal{F}_j$, with each vector component representing the relevance of a CC in F_j.

The algorithm comprises three main steps: (i) initialization, (ii) inverse family frequency computation, and (iii) CC frequency computation. First, we extract the

	$\mathcal{F}_1$			$\mathcal{F}_2$		$\mathcal{F}_3$	$\mathcal{F}_4$
Apps	app_1	app_2	app_3	app_4	app_5	app_6	app_7
Is c_i in app_k?	✓	×	✓	✓	×	×	×
freq(c_i, app_k)	5	0	4	1	0	0	0
$\texttt{ccf}(c_i, \mathcal{F}_j)$	$9/5$			$1/1$		0	0
$\texttt{iff}(c_i, \mathcal{M}_1)$	$\log \frac{4}{1+2} = 0.288$						
$I(c_i, \mathcal{F}_j, \mathcal{M}_1)$	0.518			0.288		0.000	0.000

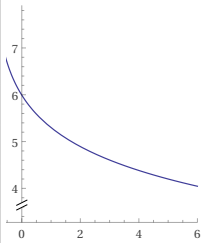
(a) Rather common CC with $|\mathcal{M}_1| = 4$.



(b) $\texttt{iff}(c_i, \mathcal{M}_1)$

	$\mathcal{F}_1$			$\mathcal{F}_2$		$\cdots$	$\mathcal{F}_{400}$
Apps	app_1	app_2	app_3	app_4	app_5	$\cdots$	app_n
Is c_i in app_k?	✓	×	×	✓	✓	×	✓
freq(c_i, app_k)	5	0	0	2	7	0	1
$\texttt{ccf}(c_i, \mathcal{F}_j)$	$5/5$			$9/7$		0	$1/1$
$\texttt{iff}(c_i, \mathcal{M}_2)$	$\log \frac{400}{1+3} = 4.605$						
$I(c_i, \mathcal{F}_j, \mathcal{M}_2)$	4.605			5.921		0.000	4.605

(c) Very uncommon CC with $|\mathcal{M}_2| = 400$.



(d) $\texttt{iff}(c_i, \mathcal{M}_2)$

Figure 4.4: Computation of $I(c_i, \mathcal{F}_j, \mathcal{M})$ and distribution of the `iff` value depending on the popularity of the CC in two different malware datasets: *tiny* (a) and (b), and *large* (c) and (d). Figure (b) and (d) represents the resulting `iff` with respect to the `FCC`, i.e.: $\texttt{iff}(c_i, \mathcal{M}) = log(\frac{|\mathcal{M}|}{x})$, where $x = 1 + |\{\mathcal{F}_i \in \mathcal{M}_2 \ : \ c \in \texttt{FCC}(\mathcal{F}_j)\}|$ and $x = 1 + |\{\mathcal{F}_i \in \mathcal{M}_2 \ : \ c \in \texttt{FCC}(\mathcal{F}_j)\}|$ respectively.

frequency $\texttt{freq}(\texttt{c}, \texttt{app})$ for every CC $c \in \texttt{CC}(app)$ of each app $app \in \mathcal{M}$ (lines 2–5). The inverse family frequency is then computed for each extracted CC using Eq. (4.8) (lines 8–10). Finally, the frequency of each CC is computed by applying Eq. (4.8), and the associated indicator for the CC is obtained (lines 11–16).

4.4.4 Modeling Families and Classifying Malware Instances

In our first experiment, we have tested the ability to correctly predict the family of a malware instance. To do this, we have randomly split our dataset into k complementary folds, being $k = 10$. During the generation of each fold, we have guaranteed that

Algorithm 1. Computing Family Vectors

Input:

Dataset of labeled malware apps (sequences of code chunks):

$\mathcal{M} = \{(app_1, \mathcal{F}_{app_1}), (app_2, \mathcal{F}_{app_2}), \ldots, ((app_p, \mathcal{F}_{app_m}))\}$

where $\mathcal{F}_{app_i} \in \{\mathcal{F}_1, \ldots, \mathcal{F}_q\}$

Output:

Vectors $\mathbf{v_j} = (l_{1,j}, \ldots, l_{k,j})$ for each $\mathcal{F}_j \in \{\mathcal{F}_1, \ldots, \mathcal{F}_q\}$

Algorithm:

1 $\texttt{FCC}(\mathcal{F}_j) = \emptyset \ \forall j = 1, \ldots, q$
2 **For each** $(app, \mathcal{F}_{app}) \in \mathcal{M}$ **do**
3 $\quad \texttt{FCC}(\mathcal{F}_a pp) = \texttt{FCC}(\mathcal{F}_a pp) \cup \texttt{CC}(app)$
4 $\quad$ Update $\texttt{freq}(c, app)$ for each $c \in \texttt{CC}(app)$
5 **end-for**
6 $\texttt{C}(\mathcal{M}) = \bigcup_{j=1}^{q} \texttt{FCC}(\mathcal{F}_j)$
7 $k = |\texttt{C}(\mathcal{M})|$
8 **For each** $i = 1, \ldots, k$ **do**
9 $\quad$ Compute $\texttt{iff}(c_i, \mathcal{M})$ according to (4.8)
10 **end-for**
11 **For each** $\mathcal{F}_j$ **do**
12 $\quad$ **For each** $i = 1, \ldots, k$ **do**
13 $\quad\quad$ Compute $\texttt{ccf}(c_i, \mathcal{F}_j)$ according to (4.7)
14 $\quad\quad$ $\mathbf{v_j}[i] = l_{i,j} = \texttt{ccf}(c_i, \mathcal{F}_j) \cdot \texttt{iff}(c_i, \mathcal{M})$
15 $\quad$ **end-for**
16 **end-for**
17 **return** $\{\mathbf{v_1}, \ldots, \mathbf{v_q}\}$

Figure 4.5: Algorithm for obtaining each family vector.

every family-subset contains at least one sample. Once the dataset is partitioned, we have randomly selected one fold as validation data, and the remaining ones are used as training data.

The training folds were used to derive a vectorial representation for each malware family as described in Section 4.4.1. A total number of 84854 CC were found across all instances in the dataset, so each family is represented by a vector with this dimensionality, as specified in (4.6). We note, however, that such vectors are

Algorithm 2. 1-NN Malware Classifier

Input:
Family vectors $\{\mathbf{v_1}, \ldots, \mathbf{v_q}\}$ and data structures $\langle \mathtt{C}(\mathcal{M}), \mathtt{iff}(c_i, \mathcal{M})\rangle$
Malware instance *app*

Output:
Predicted family $\mathcal{F}_j$

Algorithm:
1 **for each** $c_i \in \mathtt{C}(\mathcal{M})$ **do**
3 $\quad \mathbf{z}[i]) = \mathtt{freq}(c_i, app) \cdot \mathtt{iff}(c_i, \mathcal{M})$
4 **end-for**
5 $j = \arg\ \min_i\{\mathsf{dist}(\mathbf{z}, \mathbf{v_i})\}$
6 **return** $\mathcal{F}_j$

Figure 4.6: 1-NN malware classification algorithm.

very sparse (as expected by the analysis given in Section 4.3), which in practice makes it very efficient to store and manipulate them. For illustration purposes, the largest family vectors correspond to `DroidKungFu3` (19091 non-null components), `AnserverBot` (17634), `Pjapps` (15127), and `Geinimi` (12140). On average, only around 11% of each feature vector contains discriminant information.

The validation fold was processed in a similar way, obtaining a vectorial representation for each malware instance. We then implemented a 1-NN (nearest neighbor) classifier [Tan, 2005] to compute the predicted family for each malware instance under test. Such a prediction is the family whose vector is closest to the instance's vector (see Fig. 4.6). 1-NN is a widely used method in data mining that only requires to compute n distances and one minimum. To compute distances between vectors, we relied on the well-known cosine similarity:

Definition 8 (Cosine similarity)**.** *The cosine similarity between two vectors* $\mathbf{z} =$

$(z_1, \ldots, z_k)$ *and* $\mathbf{v} = (v_1, \ldots, v_k)$ *is given by*

$$\text{sim}(\mathbf{z}, \mathbf{v}) = \cos(\theta_{\mathbf{z},\mathbf{v}}) = \frac{\mathbf{z} \cdot \mathbf{v}}{\| \mathbf{z} \| \| \mathbf{v} \|} = \frac{\sum_{i=1}^{k} u_i v_i}{\sqrt{\sum_{i=1}^{k} u_i^2} \sqrt{\sum_{i=1}^{k} v_i^2}}. \tag{4.9}$$

The cosine similarity, which measures the cosine of the angle between vectors $\mathbf{z}$ and $\mathbf{v}$, has been extensively used to compare documents in text mining and information retrieval applications. Besides, it is quite efficient to evaluate in domains such as ours, since vectors are sparse and, therefore, only a few non-zero dimensions need to be considered in the computation. As for our purposes a distance, and not a similarity, is required, we use:

$$\text{dist}(\mathbf{z}, \mathbf{v}) = 1 - \text{sim}(\mathbf{z}, \mathbf{v}). \tag{4.10}$$

Results have been cross-validated with each of the k folds that were previously generated. The classification error per family attained in this experiment is shown in Table 4.2. A closer inspection reveals that the classification error is not uniform across families. On the contrary, errors concentrate on 6 out of the 33 malware families studied (`AnserverBot`, `BaseBridge`, and `DroidKungFu1` through `DroidKungFu4`), while instances belonging to the remaining 27 families are perfectly classified.

Interestingly, `DroidKungFu` has been considered a milestone in Android OS malware sophistication [Zhou and Jiang, 2012]. After the release of its first version, a number of variants rapidly emerged, including `DroidKungFu2` through `DroidKungFu4`

Classification Error (%)			
ADRD	0.00%	GingerMaster	0.00%
AnserverBot	4.66%	GoldDream	0.00%
Asroot	0.00%	Gone60	0.00%
BaseBridge	7.92%	HippoSMS	0.00%
BeanBot	0.00%	KMin	0.00%
Bgserv	0.00%	NickySpy	0.00%
CruseWin	0.00%	Pjapps	0.00%
DroidDream	0.00%	Plankton	0.00%
DroidDreamLight	0.00%	RogueLemon	0.00%
DroidKungFu1	12.92%	RogueSPPush	0.00%
DroidKungFu2	19.46%	SndApps	0.00%
DroidKungFu3	8.12%	Tapsnake	0.00%
DroidKungFu4	18.21%	YZHC	0.00%
DroidKungFuSapp	0.00%	Zsone	0.00%
FakePlayer	0.00%	jSMSHider	0.00%
GPSSMSSpy	0.00%	zHash	0.00%
Geinimi	0.00%		

Table 4.2: Average malware classification error per family using 1-NN with 10-fold cross-validation.

or `DroidKungFuApp`. A common feature shared by all these variants is the use of encryption to hide their existence. In fact, some of them embedded their payloads within constant strings or even resource files (e.g., pictures, asset files, etc.). Furthermore, `DroidKungFu` aggressively obfuscates the class name and uses native programs (Java Native Interface, or JNI) precisely to made the analysis difficult. Similarly, `AnserverBot` use sophisticated techniques to obfuscate all internal classes, methods, and fields. Moreover, instead of enclosing the payload within the app, `AnserverBot` dynamically fetches and loads it at runtime (this is known as *update attacks*). In this regard, some authors (e.g., [Zhou and Jiang, 2012]) believe that `AnserverBot` actually evolved from `BaseBridge` and inherited this feature from it. Our results seem to confirm this hypothesis.

More insights can be gained by observing the confusion matrix given by a larger dataset of $k = 2$ folds and approximately an equal number of malware instances in each fold (see Table 4.3). Each cell (x, y) in the matrix shows the number of instances belonging to family x whose predicted family is y. Here, for instance, we can observe that 5 out of the 61 samples of `BaseBridge` have been predicted as `AnserverBot`. Similarly, we can observe that a few samples of `DroidKungFu1` have been classified as `DroidKungFu2` and, in a similar way, there is some misclassifications between `DroidKungFu3` and `DroidKungFu4`. Thus, the aforementioned classification error is actually justified by the evolutionary relationships of these particular malware strands.

4.4.5 Evolutionary Analysis of Malware Families

In this section, we discuss the application of hierarchical clustering to the feature vectors that model samples and family. The resulting dendrograms are then used to conjecture about their evolutionary phylogenesis, giving a valuable instrument to discover relationships among families. We first describe the hierarchical clustering algorithm currently included in Dendroid. Subsequently we discuss the results obtained.

4.4.5.1 Single Linkage Hierarchical Clustering

Single Linkage Clustering, also known as nearest neighbor clustering, is a well-known method to carry out an agglomerative hierarchical clustering process over a population of vectors. The algorithm, shown in Figure 4.7, keeps a set of clusters, $\mathcal{K}$, which is initialized to the set of family vectors. At each iteration it, the two closest clusters

Actual \ Predicted	ADRD	AnserverBot	Asroot	BaseBridge	BeanBot	Bgserv	CruseWin	DroidDream	DroidDreamLight	DroidKungFu1	DroidKungFu2	DroidKungFu3	DroidKungFu4	DroidKungFuSapp	FakePlayer	GPSSMSSpy	Geinimi	GingerMaster	GoldDream	Gone60	HippoSMS	KMin	NickySpy	Pjapps	Plankton	RogueLemon	RogueSPPush	SndApps	Tapsnake	YZHC	Zsone	jSMSHider	zHash	
ADRD	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**11**
AnserverBot	0	89	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**93**
Asroot	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**4**
BaseBridge	0	5	0	56	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**61**
BeanBot	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**4**
Bgserv	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**4**
CruseWin	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
DroidDream	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**8**
DroidDreamLight	0	0	0	0	0	0	0	0	23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**23**
DroidKungFu1	0	0	0	0	0	0	0	0	0	15	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**17**
DroidKungFu2	0	0	0	0	0	0	0	0	0	3	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**15**
DroidKungFu3	0	0	0	0	0	0	0	0	0	0	0	141	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**154**
DroidKungFu4	0	0	0	0	0	0	0	0	0	0	0	8	40	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**48**
DroidKungFuSapp	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
FakePlayer	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
GPSSMSSpy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
Geinimi	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	34	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**34**
GingerMaster	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
GoldDream	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**23**
Gone60	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	**4**
HippoSMS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	**2**
KMin	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	26	0	0	0	0	0	0	0	0	0	0	0	**26**
NickySpy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	**1**
Pjapps	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	22	0	0	0	0	0	0	0	0	0	**22**
Plankton	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	**5**
RogueLemon	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	**1**
RogueSPPush	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	**4**
SndApps	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	**5**
Tapsnake	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**1**
YZHC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	0	0	0	**11**
Zsone	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	0	0	**6**
jSMSHider	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	0	**8**
zHash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	**5**
	11	**94**	**4**	**60**	**4**	**4**	**1**	**8**	**23**	**18**	**14**	**149**	**53**	**1**	**3**	**3**	**34**	**2**	**23**	**4**	**2**	**26**	**1**	**22**	**5**	**1**	**4**	**5**	**1**	**11**	**6**	**8**	**5**	**610**

Table 4.3: Confusion matrix for malicious app classification.

$\mathbf{r}, \mathbf{s} \in \mathcal{K}$ are combined into a larger cluster $\mathbf{v_{rs}}$. The distance matrix between each pair of clusters is then updated by removing both $\mathbf{r}$ and $\mathbf{s}$, adding the newly created $\mathbf{v_{rs}}$, and finally computing the distances from $\mathbf{v_{rs}}$ to each remaining cluster $\mathbf{x}$ through a linkage function. In our case, such a function is simply the shortest between the distance from $\mathbf{x}$ to $\mathbf{r}$ and the distance $\mathbf{x}$ to $\mathbf{s}$. Furthermore, the algorithm keeps a list $L(it)$ with the distances at which each fusion takes place. The process is iterated until the set of clusters $\mathcal{K}$ is reduced to one element.

Algorithm 3. Single-linkage hierarchical clustering of malware families

Input:

Family vectors $\{\mathbf{v_1}, \ldots, \mathbf{v_q}\}$

Output:

Proximity matrices $D^{(t)} = [d_{ij}]$ and linkages at each level $L(k)$

Algorithm:

1 $\mathcal{K} = \{\mathbf{v_1}, \ldots, \mathbf{v_q}\}$
2 $D^{(0)} = [d_{ij}] = \text{dist}(\mathbf{v_i}, \mathbf{v_j})$ for all $\mathbf{v_i}, \mathbf{v_j} \in \mathcal{K}$
3 $it = 0, L(it) = 0$
4 **while** $|\mathcal{K}| \neq 1$ **do**
5 Find $\mathbf{r}, \mathbf{s} \in \mathcal{K}$ such that $\text{dist}(\mathbf{r}, \mathbf{s}) = \min_{\mathbf{a},\mathbf{b} \in \mathcal{K}} \{\text{dist}(\mathbf{a}, \mathbf{b})\}$
6 Merge $\mathbf{r}, \mathbf{s}$ into new cluster $\mathbf{v_{rs}}$
7 $it = it + 1$
8 $L(it) = \text{dist}(\mathbf{r}, \mathbf{s})$
9 $D^{(it)} = D^{(it-1)}$ deleting the rows and columns corresponding to $\mathbf{r}$ and $\mathbf{s}$
10 Add to D a new row and column for $\mathbf{v_{rs}}$
11 $D[\mathbf{v_{rs}}, \mathbf{x}] = \text{dist}(\mathbf{v_{rs}}, \mathbf{x}) = \min\{\text{dist}(\mathbf{r}, \mathbf{x}), \text{dist}(\mathbf{s}, \mathbf{x})\}$ for all $\mathbf{x}$
12 $\mathcal{K} = \mathcal{K} \cup \{\mathbf{v_{rs}}\} \setminus \{\mathbf{r}, \mathbf{s}\}$
13 **end-while**
14 **return** $\langle D^{(0)}, \ldots, D^{(it)}, L \rangle$

Figure 4.7: Single linkage hierarchical clustering algorithm for malware families.

4.4.5.2 Results and Discussion

The results of a hierarchical clustering can be visualized in a dendrogram as the one depicted in Figure 4.9 for the dataset used in this work. The dendrogram represents a tree diagram where links between the leaves (malware families) illustrate the parental relationships (ancestors and descendants) in a hierarchy. Thus, clusters (denoted as v_{rs} in Figure 4.7–line 6) are tree nodes representing merged families, i.e., a common ancestor. The paths that group together different families illustrate the phylogenetic evolution of the "species." Furthermore, the distance $\mathcal{D}^{(t)}$ between an ancestor and its descendants is a measure of their similarity and, therefore, can be interpreted as an

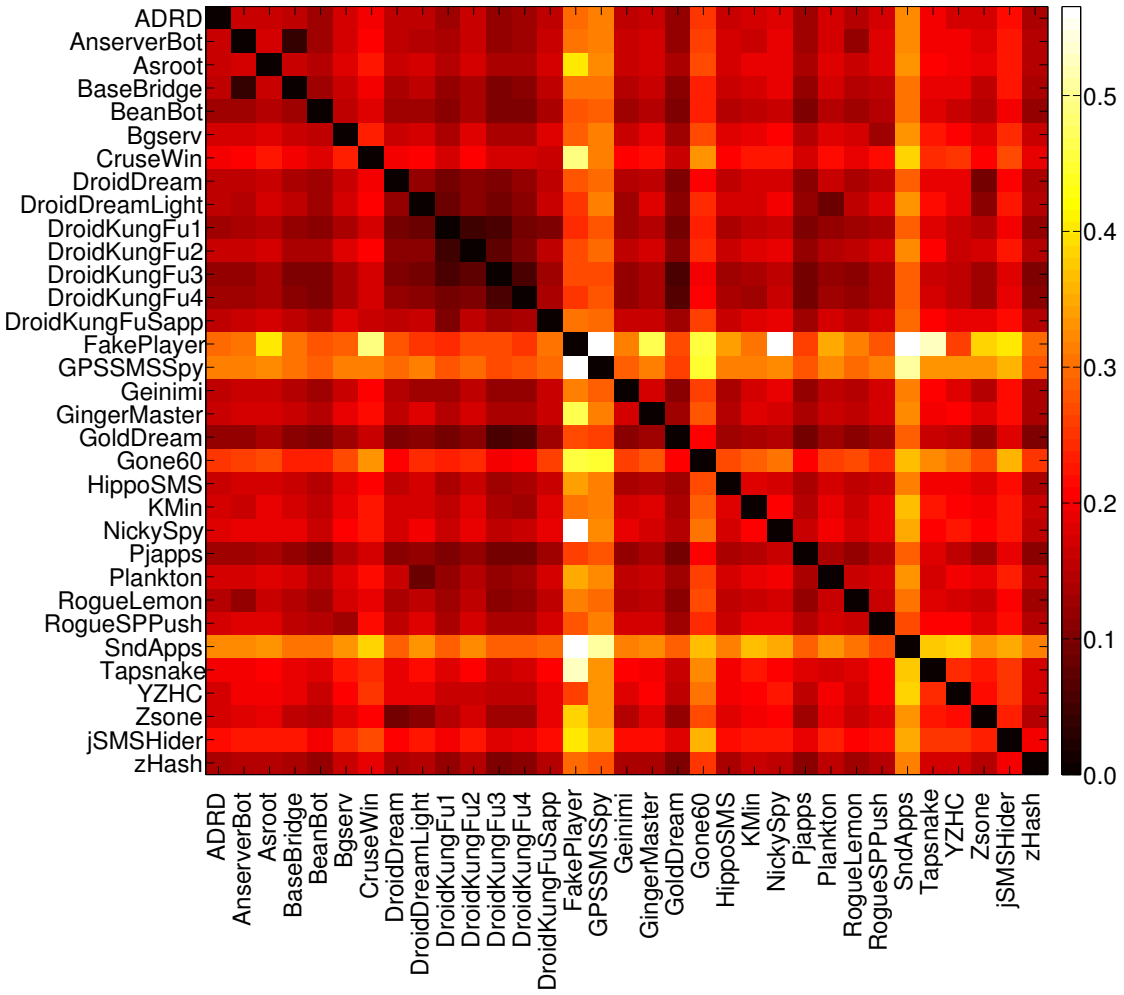


Figure 4.8: Distance matrix between pairs of malware families.

evolutionary (or diversification) distance. Note that the sequence of such distances is provided as an output by the algorithm in Figure 4.7.

The initial proximity matrix, $\mathcal{D}^{(0)}$, for all the families in our dataset is graphically shown in Figure 4.8. As anticipated by the results of the previous experiment, the similarity among some groups of families is striking, while in other cases there are substantial differences. The results after applying hierarchical clustering to the datasets are displayed in the dendrogram shown in Figure 4.9. There are a number

of interesting observations:

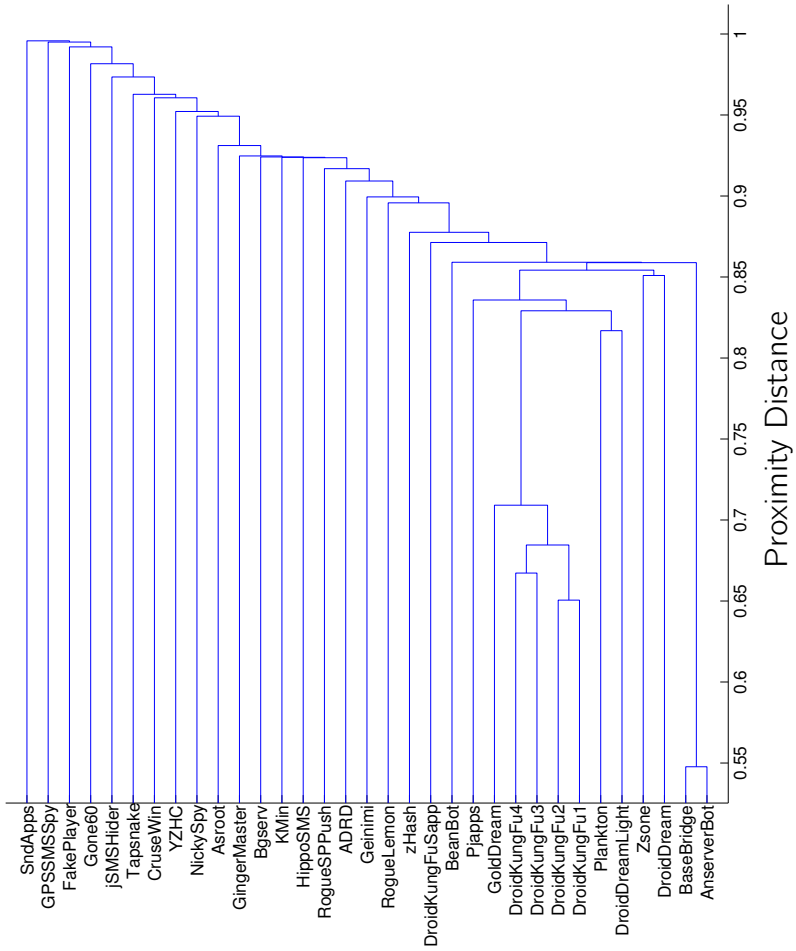


Figure 4.9: Dendrogram obtained after hierarchical clustering over the dataset.

- `BaseBridge` and `AnserverBot` are intimately related, hence that they appear as variants of a common ancestor. Besides, their linkage (distance) is very small compared to the rest of the families, which suggest a large share of relevant code structures and, perhaps of functionality too.

- The case of the `DroidKungFu` variants is remarkably captured. It transpires from our results that `DroidKungFu1` and `DroidKungFu2` are alike, and the same occurs with the pair `DroidKungFu3` and `DroidKungFu4`. Furthermore,

both pairs descend from a common ancestor, say `DroidKungFuX`, which in turn is connected with `GoldDream`. This branch connects with another one formed by the pair `Plankton-DroidDreamLight`, and both groups relate to `Pjapps`, which is among the oldest examples of sophisticated Android OS malware. Finally, the relationship between this group, `Zsone-DroidDream`, and `BaseBridge-AnserverBot` could be explained by a number of reasons, including the fact that they probably share common engines.

- The remaining malware families seem rather unrelated, and no significant evolutionary relationship can be inferred. Note, too, that distances approach 1 in this area of the dendrogram, which suggest a very weak connection.

4.5 Conclusions

In this chapter, we have proposed a text mining approach to automatically classify smartphone malware samples and analyze families based on the code structures found in them. Our proposal is supported by a statistical analysis of the distribution of such structures over a large dataset of real examples. Our findings point out that the problem bears strong resemblances to some questions arising in automated text classification and other information retrieval tasks. By adapting the standard Vector Space Model commonly used in these domains, we have explored the suitability of such techniques to measure similarity among malware samples, and to classify unknown samples into known families. Our experimental results suggest that this technique is fast, scalable and very accurate. We have subsequently studied the use of hierarchical clustering to derive dendrograms that can be understood as phylogenetic trees for malware families. This provides the analyst with a means to

analyze the relationships among families, the existence of common ancestors, the prevalence and/or extinction of certain code features, etc. As discussed in this Thesis, automated tools such as these will be instrumental for analysts to cope with the proliferation and increasing sophistication of malware.

III

Dynamic-based Analysis

5

Alterdroid: Differential Fault Analysis of Obfuscated Malware Behavior

5.1 Introduction

More sophisticated obfuscation techniques, particularly in code, are starting to materialize [Huang et al., 2013; Linn and Debray, 2003; Rastogi et al., 2013b]. These techniques and trends create an additional obstacle to malware analysts, who see their task further complicated and have to ultimately rely on carefully controlled dynamic analysis techniques to detect the presence of potentially dangerous pieces of code.

Approaches based on dynamic code analysis such as the ones described in Chapter 2 are promising, but current works [Egele et al., 2012], [Rastogi et al., 2013a], [Shabtai et al., 2014] only provide an holistic understanding of the behavior of an app. This feature challenges the identification of grayware and the attribution of malicious behavior to components of the app. Thus, current approaches are prone to miss on their identification and further human—costly—efforts are required as

shown by Zhou and Jiang in [Zhou and Jiang, 2012].

Recent works approach the detection of obfuscated malware by mining identifiable static features such as cryptographic functions [Calvet et al., 2012]. However, Schrittwieser et al. [Schrittwieser et al., 2013] demonstrate the incompleteness of these and other semantic-aware detectors [Christodorescu et al., 2005] by means of "covert computation". As regards the various ways to obfuscate or locate obfuscated code in binary data, [Ker et al., 2013] describes most relevant steganography and steganalysis techniques including active [Fisk et al., 2003; Li and Craver, 2011] and passive wardens.

Fuzz Testing or Fuzzing is a technique commonly used for providing inputs when testing software for security purposes [Takanen et al., 2008]. Fuzzing technique is recently gaining popularity for automating the dynamic analysis of apps in smartphones [Machiry et al., 2013; Mahmood et al., 2012; Rastogi et al., 2013a; Shabtai et al., 2014; Zheng et al., 2012]. Basically, Fuzzing aims at providing different streams of events to the app for further monitoring the behavior of the device. Fuzzing was originally proposed for finding software crashes or unexpected behaviors by deliberately introducing faulty inputs.

In this chapter we describe Alterdroid [Suarez-Tangil et al., 2014a], a fuzzing-based technique for detecting obfuscated malware components distributed as parts of an app package. Such components are often hidden outside the app main code components, as these may be subject to static analysis by market operators. The key idea in Alterdroid consists of analyzing the behavioral differences between the original app and an altered version where a number of modifications (*faults*) have been carefully introduced. Such modifications are designed to have no observable

effect on the app execution, provided that the altered component is actually what it should be and does not have any hidden functionality. For example, replacing the value of some pixels in a picture or a few characters in a string encoding an error message should not affect execution. However, if after doing so it is observed that a dynamic class loading action crashes or a network connection does not take place, it may well be that the picture was actually a piece of code or the string a network address or a URL.

At high level, Alterdroid has two differentiated major components: fault injection and differential analysis. The first one takes a candidate app—the entire package—as input and generates a fault-injected one. This is done by first extracting all components in the app and then identifying those suspicious of containing obfuscated functionality. Such identification is done on an anomaly-detection basis by comparing certain statistical features of the component's contents with a predefined model for each possible type of resource (i.e., code, pictures and video, text files, databases, etc.). Faults are then injected into candidate components, which are subsequently repackaged, together with the unaltered ones, into a new app. This process admits simultaneous injection of different faults into different components and is driven by a search algorithm that attempts to identify where the obfuscated functionality is hidden. Both the original and the fault-injected apps are then executed under identical conditions (i.e., context and user inputs), and their behavior is monitored and recorded in the form of two activity signatures. Such signatures are merely sequential traces of the activities executed by the app, such as for example opening a network connection, sending or receiving data, loading a dynamic component, sending an sms, interacting with the file system, etc. Both behavioral signatures are then treated as in a string-to-string correction problem, in such a way that computing

the Levenshtein distance (also known as edit distance) between them returns the list of observable differences in terms of insertions, deletions and substitutions. Such a list, called the differential signature, is finally matched against a rule set where each rule encodes a relationship between the type of functionality presumably hidden and certain patterns in the differential signature.

The contributions of this chapter can be summarized in what follows:

1. We introduce the notion of differential fault analysis for detecting obfuscated malware functionality in smartphone apps.

2. We provide simple yet powerful theoretical models for fault injection operators, behavioral signatures and rule-based analysis of differential behavior.

3. We describe the functional components of Alterdroid, a prototype implementation of our differential fault analysis model for Android apps. The system includes instantiations for key tasks such as identifying components to be fault-injected and a search-based approach to track down obfuscated components in an app.

4. We also show how Alterdroid's functional architecture supports a distributed deployment of different modules, which allows offloading various analysis to the cloud and running them in parallel.

Additionally, we illustrate our approach by discussing the step-by-step analysis of three Android malware samples that incorporate hidden functionality in repackaged apps: `DroidKungFu`, `AnserverBot`, and `GingerMaster`.

Fault injection analysis has been widely used for software assurance against fault tolerance [Gray, 1986; Natella et al., 2013]. Our approach uses Fuzzing both for au-

tomating the generation of inputs given to the sandbox and to inject fault conditions into components of the program. To the best of our knowledge differential fault analysis is a novel approach compared to existing works aiming at analyzing malware in smartphones.

The rest of this chapter is organized as follows. In Section 5.2 we introduce formal models for fault injection and differential analysis. Section 5.3 describes Alterdroid's architecture and its key functional components, discusses the complexity of differential fault analysis, and provides an overview of our proof-of-concept implementation. Subsequently in Section 5.4 we describe the analysis of three Android malware samples with Alterdroid. Finally, in Section 6.6 we conclude the chapter by summarizing our main contributions and discussing limitations and directions for future research.

5.2 A Differential Fault Analysis Model

This section introduces the theoretical background used in Alterdroid [Suarez-Tangil et al., 2014a] to inject faults into apps, represent behavioral differences between apps, and deducing properties from such behavioral differences considering injected faults and observed differences. The overall dynamics of the differential fault analysis process (i.e., the mechanism governing which faults are injected and where) is external to this model and will be discussed later in Section 5.3.

5.2.1 Fault Injection Model

An app $\mathcal{P}$ can be seen as a collection of components

$$\mathcal{P} = \{c_1, c_2, \ldots, c_k\}. \tag{5.1}$$

A component can be composed of a number of classes (i.e., code), but also other resources that are dynamically accessed, such as for example asset files. Components have a type, such as for example code, picture, video, database, etc. We define a type function $\tau(c)$ that returns the type of component c.

Fault conditions can be injected into an app by altering one or more of its components. Assume that $\mathbb{C}$ is the set of all possible app components and $\psi(c)$ is the alteration made over a component $c \in \mathbb{C}$. A Fault Injection Operator is a transformation

$$\begin{gathered} \psi^{c_i} : 2^{\mathcal{P}} \rightarrow 2^{\mathbb{C}} \\ \psi^{c_i}(\mathcal{P}) = \mathcal{P} \setminus \{c_i\} \cup \{\psi(c_i)\} \end{gathered} \tag{5.2}$$

That is, $\psi^{c_i}(\mathcal{P})$ returns a modified version of $\mathcal{P}$ where component c_i has been replaced by $\psi(c_i)$. Depending on the functionality of c and the nature of the modifications introduced by ψ, replacing c by $\psi(c)$ may, or may not, translate into observable differences in the execution of $\mathcal{P}$. In this work, we restrict ourselves to FIOs that make alterations to data components only, not to instructions. Data components include the value of variables found in the code, and also asset files such as databases, pictures, audio and video files, etc. We will abuse notation and write $\tau(\psi^{c_i})$ for $\tau(c_i)$; i.e., we consider that the type of a FIO is the type of all components it can be applied to.

FIOs can be arbitrarily complex and, in some cases, their operation may depend

on the type and/or current value of the component to be altered. We will also make use of very simple FIOs that treat components as a bit string, such as for example:

- $\mathsf{rrep}^c(\cdot)$: replaces the value of component c for a randomly chosen bit string.
- $\mathsf{zero}^c(\cdot)$: replaces the value of component c for a string of zeros of the same length.
- $\mathsf{rmut}^c_j(\cdot)$: flips the j-th bit of of component c.

The FIOs defined above are rather generic. In some cases, we might want to define further datatype-specific operators. These will be useful to modify in a syntax-preserving way certain data objects (e.g., multimedia files) when the focus is on changing the content without rendering the object unusable.

5.2.2 Modeling Differential Behavior

A key task in our system is the analysis of the behavioral differences between an original app and a slightly modified version of it after applying a FIO. We next introduce a model to represent traces of activities and differences between traces.

5.2.2.1 Behavioral Signatures

An app interacts with the platform where it is executed by requesting services through a number of available system calls. These define an interface for apps that need to read/write files, send/receive data through the network, make a phone call, etc. Rather than focusing on low-level system calls, in this work we will describe an app's behavior through the *activities* it executes (see also Chapter 7–Section 7.2.3).

In some cases there will be a one-to-one correspondence between an activity and a system call, while in others an activity encompasses a sequence of system calls executed in a given order. In what follows, we assume that

$$\mathbb{A} = \{a_1, a_2, \ldots, a_r\} \tag{5.3}$$

is a set of all relevant and observable activities an app can execute.

The execution flow of an app $\mathcal{P}$ may follow different paths depending on its inputs. We group such inputs into two main classes:

- A sequence $\mathbf{u}$ of user-provided inputs U, such as for example those acquired through the touchscreen.
- A sequence of contexts $\mathbf{g}$, defining the state of the environment when the execution takes place. Each context (state) is represented by a set of variables that provide the app with information such as current location, time, energy level, temperature, etc.

We will denote by $\mathcal{P}(\mathbf{u}|\mathbf{g})$ the execution of $\mathcal{P}$ with user inputs $\mathbf{u}$ in context $\mathbf{g}$.

The observable behavior resulting from the execution of $\mathcal{P}(\mathbf{u}|\mathbf{g})$ is summarized in a *behavioral signature* $\beta[\mathcal{P}(\mathbf{u}|\mathbf{g})]$, this being a time series given by

$$\beta[\mathcal{P}(\mathbf{u}|\mathbf{g})] = \langle s_1, s_2, \ldots, s_n \rangle, \quad s_i \in \mathbb{A} \tag{5.4}$$

Note that this signature model does not take into account the duration of each activity or the time elapsed between each two of them, but only their relative order. We will abuse notation and omit the associated app and its inputs when it is irrelevant

or clear from context.

Finally, we will denote by $\text{len}(\beta)$ the length of signature β, defined as the number of activities in the series.

5.2.2.2 Differential Signatures

We are interested in analyzing the differences between two observed behaviors given by their respective behavioral signatures. We approach this problem as one of string-to-string correction, where differences are represented as the minimum number of edit operations needed to transform one signature into the other. Given an behavioral signature $\beta = \langle s_1, s_2, \dots, s_n \rangle$, we define the next three families of signature transformation operators (STO):

- $\text{Ins}_i^a(\beta) = \langle s_1, \dots, s_i, a, s_{i+1}, \dots, s_n \rangle \quad \forall a \in \mathbb{A}, \quad \forall i \in [1, n]$
- $\text{Del}_i(\beta) = \langle s_1, \dots, s_{i-1}, s_{i+1}, \dots, s_n \rangle \quad \forall s_i \in \mathbb{A}, \quad \forall i \in [1, n]$
- $\text{Sub}_i^a(\beta) = \langle s_1, \dots, s_{i-1}, a, s_{i+1}, \dots, s_n \rangle \quad \forall a \in \mathbb{A}, \quad \forall i \in [1, n]$

Let

$$\mathbb{O} = \bigcup_{i,a} \left(\text{Ins}_i^a \cup \text{Del}_i \cup \text{Sub}_i^a \right) \tag{5.5}$$

be the set of all possible STOs. Given two behavioral signatures β_1 and β_2, we define the *differential signature* $\Delta(\beta_1, \beta_2)$ as an ordered sequence of STOs:

$$\Delta(\beta_1, \beta_2) = \langle o_1, o_2, \dots, o_k \rangle \quad o_i \in \mathbb{O} \tag{5.6}$$

such that

$$o_k \circ o_{k-1} \circ \cdots \circ o_1(\beta_1) = \beta_2 \tag{5.7}$$

where $o_i \circ o_j$ denotes the composition of STOs o_i and o_j. In other words, the differential signature $\Delta(\beta_1, \beta_2)$ provides a sequence of insertions, deletions and substitutions that transforms β_1 into β_2. Note that, in general, $\Delta(\beta_1, \beta_2) \neq \Delta(\beta_2, \beta_1)$.

For the purposes of this work, we are interested in minimal differential signatures, i.e., sequences of minimum length. The most straightforward way to compute the minimal differential signature is by computing the Levenshtein distance (also known as edit distance) between β_1 and β_2 assuming that all operators have equal cost [Kumazawa and Tamai, 2011]. This computation returns not only the distance, but also the optimal differential signature.

5.2.3 Analyzing Differential Signatures

Let

$$\mathcal{P}' = \Psi(\mathcal{P}) = \psi_r^{c_r} \circ \psi_{r-1}^{c_{r-1}} \circ \cdots \circ \psi_1^{c_1}(\mathcal{P}) \tag{5.8}$$

be the app resulting after the sequential application of FIOs $\psi_1, \ldots, \psi_r$ to components $c_1, \ldots, c_r$ of app $\mathcal{P}$. Let $\beta[\mathcal{P}]$ and $\beta[\Psi(\mathcal{P})]$ be the behavioral signatures obtained after executing $\mathcal{P}$ and $\Psi(\mathcal{P})$ under the same conditions[1], and let $\Delta(\beta[\mathcal{P}], \beta[\Psi(\mathcal{P})])$ be their differential signature. The analysis model used in this work is based on deducing properties of $\mathcal{P}$ from the presence or absence of certain *patterns* in $\Delta(\beta[\mathcal{P}], \beta[\Psi(\mathcal{P})])$ and the properties of the FIO Ψ. We next describe these two elements in turn.

Note than ψ denotes a single FIO operating on a given component and Ψ a number of of FIOs operating on a collection of components of an app $\mathcal{P}$.

[1] That is, the same sequence of user inputs and contexts.

5.2.3.1 FIO Classes

We identify two broad classes of FIOs:

- A FIO ψ^{c_i} is said to be *indistinguishable* if $\Delta(\beta[\mathcal{P}], \beta[\psi^{c_i}(\mathcal{P})]) = \emptyset$ for all apps P containing component c_i. In other words, a FIO is indistinguishable if it does not affect the execution flow of any app and, therefore, the behavioral signatures before and after applying it coincide.

- A FIO ψ^{c_i} is said to be *distinguishable* if $\Delta(\beta[\mathcal{P}], \beta[\psi^{c_i}(\mathcal{P})]) \neq \emptyset$ for all apps P containing component c_i. Thus, distinguishable FIOs always manifest as nonempty differential signatures.

In what follows, the predicate $\mathsf{ind}(\psi^{c_i})$ models this property:

$$\mathsf{ind}(\psi^{c_i}) = \begin{cases} \texttt{true} & \text{if } \psi^{c_i} \text{ is indistinguishable} \\ \texttt{false} & \text{otherwise} \end{cases} \tag{5.9}$$

5.2.3.2 Properties of Differential Signatures

Patterns in differential signatures will be modeled as first-order logical predicates upon which Boolean formulae can be defined. Thus, analyzing a differential signature reduces to evaluating a number of Boolean formulae linked to properties of the app and the FIO, i.e.:

$$\mathcal{P} \text{ has property } x \iff \Phi_x\big(\Psi, \Delta(\beta[\mathcal{P}], \beta[\Psi(\mathcal{P})])\big) = \texttt{true} \tag{5.10}$$

We consider two basic predicates:

- $\mathsf{equal}(\Delta_1, \Delta_2) = \texttt{true}$ iff $\Delta_1 = \Delta_2$, where Δ_1 and Δ_2 are differential signatures. Note that the empty set is a valid differential signature.

- $\mathsf{contains}(\Delta, o) = \texttt{true}$ iff $\Delta = \langle o_1, o_2, \ldots, o_k \rangle$ and $\exists\ o_j \in \Delta$ such that $o_j = o$.

Standard symbols will be used for Boolean formulae, including quantifiers ($\exists$, $\forall$), negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), etc.

5.2.3.3 Examples

We next illustrate the concepts introduced above through a number of examples.

Example 1. Assume that c_{icon} is an icon image used by an app $\mathcal{P}$ in its user interface. Modifying some pixels of such icon, or even replacing it by another valid icon does not affect at all the execution flow of $\mathcal{P}$. If nonetheless the icon is replaced and the modified app behaves different from the original app under exactly the same conditions, it can be deduced that the original icon contained some *functionality*, such as e.g., a piece of compiled code masqueraded as an icon. This intuition can be generalized through the following rule (hidden functionality in component, or HFC):

$$\mathbf{R_{HFC}}\ :\ c \in \mathcal{P} \text{ contains hidden functionality } \iff$$
$$\mathsf{ind}(\psi^c)\ \wedge\ \neg\mathsf{equal}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \emptyset\big)$$

Example 2. A more specific case of the situation discussed above occurs when modifications on a component c result in the absence of a dynamic loading action, which are used to load code pieces into memory. In such a case, it may be possible that c contains hidden code that is dynamically loaded. The following rule captures

this:

$$\begin{aligned}&\mathbf{R_{CDC}}\ :\ c \in \mathcal{P} \text{ contains dynamic code } \iff \\ &\mathsf{ind}(\psi^c)\ \wedge\ \exists\ i\ :\ \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_i\big), s_i = \mathit{dex\_load}\end{aligned} \tag{5.11}$$

Example 3. Let v be a variable such that their content has no influence on the program flow. For example, v could be a string containing an error message which may be displayed at some point. Such strings have been broadly used in existing malware to hide URLs that point to services from where the malware can download further code, receive instructions, send data, etc. To avoid detection, the string is often obfuscated and the URL is only revealed at execution time after applying some transformations. Thus, any modification on the string such that the URL is damaged will likely result on the impossibility of establishing a connection. The following rule captures this intuition:

$$\begin{aligned}&\mathbf{R_{URL}}\ :\ v \in \mathcal{P} \text{ contains an URL } \iff \\ &\mathsf{ind}(\psi^v)\ \wedge\ \exists\ i\ :\ \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^v(P)]), \mathsf{Del}_i\big), s_i = \mathit{net}\end{aligned} \tag{5.12}$$

Example 4. Similarly to the cases discussed above, it may be possible to find out whether a component c leaks information through a number of sensors (e.g., accelerometer, GPS, etc.) if, after modifying it, the differential signature lacks an

access to such a sensor and a network connection:

$$\begin{aligned}
&\mathbf{R_{SDL}} \ : \ c \in \mathcal{P} \text{ leaks sensor data } \iff \mathsf{ind}(\psi^c) \ \wedge \\
&\Big(\exists \ i_1 \ : \ \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i_1}\big) \ \vee \\
&\quad \exists \ i_2 \ : \ \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i_2}\big) \ \vee \\
&\quad \vdots \\
&\Big) \ \wedge \ \exists \ j \ : \ \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{j}\big)
\end{aligned} \tag{5.13}$$

where $s_{i_1} = \textit{accelerometer}$, $s_{i_1} = \textit{gps}$, and $s_j = \textit{network}$

5.3 Alterdroid: Differential Fault Analysis of Obfuscated Apps

In this section we describe Alterdroid, our approach to study obfuscated malware code based on the differential fault analysis model discussed in the previous section. The high level architecture of Alterdroid is shown in Figure 5.1. There are two differentiated major blocks. The first one generates a number of fault-injected apps. This process is carried out by first extracting all app components and identifying those of interest (CoI), i.e., those suspicious of containing hidden functionality. An iterative process then selects candidate CoI and injects faults into them. Both the modified and the unmodified components are then repackaged together into a new app. The second block generates stimuli for both apps (user inputs and context) and executes them, generating a pair of behavioral signatures. The differential signature is then computed and matched against a database of patterns to identify the presence of hidden functionality.

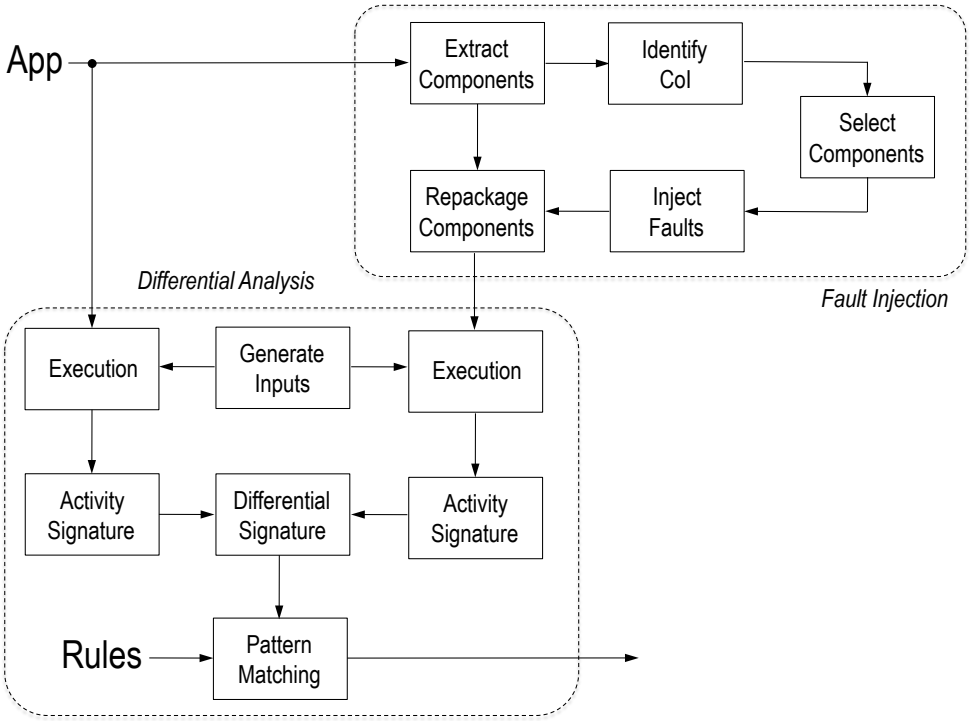


Figure 5.1: Alterdroid architecture.

We next provide a detailed description of the key modules of Alterdroid and the current prototype implementation.

5.3.1 Identifying Components of Interest

The first step in the analysis of an app is to identify *component of interest* (CoI). Such components will be later fault injected according to some strategy in order to analyze the resulting behavior.

We say that a component c of type $\tau(c)$ in an app $\mathcal{P}$ is of interest if it does not fit a model $\mathcal{M}_{\tau(c)}$ defined for all components of type $\tau(c)$. In our current version of Alterdroid, models measure statistical features only, such as for example the expected entropy, the byte distribution, or the average size. Such features are computed from

a dataset of components of the same type, such as text files, pictures, code, etc. For each model $\mathcal{M}$, we assume a Boolean function $\mathsf{test}(c, \mathcal{M})$ that returns `true` if c complies with $\mathcal{M}$, and `false` otherwise. For example, if M is a byte distribution, then $\mathsf{test}()$ could be a goodness-of-fit test (e.g., chi-square) between M and c's byte distribution. More formally,

$$c \in \mathsf{CoI}(\mathcal{P}) \iff \mathsf{test}(c, M_{\tau(c)}) = \texttt{false} \tag{5.14}$$

In our experience, such simple models suffice to spot the most common—and rather simple—obfuscation methods observed in smartphone malware, including code camouflaged as supplementary multimedia files, connection data hidden in text variables, etc.

Alterdroid also supports an exhaustive analysis mode in which some additional components may be considered CoI, even if they comply with their type model. In this mode, a component is considered CoI if it is CoI as defined above or there exists an indistinguishable operator for it. Formally

$$\begin{aligned} c \in \mathsf{CoI}(\mathcal{P}) \iff & \big(\mathsf{test}(c, M_{\tau(c)}) = \texttt{false}\big) \text{ or} \\ & \big(\exists\ \psi^c\ :\ \mathsf{ind}(\psi^c)\big) \end{aligned} \tag{5.15}$$

The rationale of this mode is to also check components for which we know in advance that alterations do not translate into noticeable differences. This is very useful to detect more sophisticated obfuscation methods that try to evade detection by carefully modifying the code so as it fits the statistical model of the component. As a side effect, however, the exhaustive analysis mode may end up with a large set of

Algorithm 1. COI Obtention

Input:
App: $\mathcal{P} = \{c_1, c_2, \ldots, c_k\}$
Set of type normality models: $\{M_1, M_2, \cdots, M_n\}$
Set of FIOs: $\{\psi_1, \psi_2, \cdots, \psi_m\}$
Mode: `normal` / `exhaustive`

Output:
CoI: List of all components of interest

Algorithm:

```
1  CoI ← ∅
2  For each c ∈ P do
3     if [test(c, M_τ(c)) = false] or
              [(mode = exhaustive) and (∃ ψ_i : τ(ψ_i) = τ(c))]
4     then
4           CoI ← CoI ∪ {c}
5  end-for
6  return CoI
```

Figure 5.2: Algorithm for obtaining components of interest from an app.

CoI.

The algorithm shown in Figure 5.2 describes the process discussed above to identify COI in Alterdroid.

5.3.2 Generating Fault-injected Apps

Components of interests identified in the previous stage are injected with faults and reassembled, together with the remaining app components, to generate a faulty app $\mathcal{P}'$. This process can generate several fault-injected apps, as there are multiple ways of applying different FIOs to different components in the CoI set. In Alterdroid, fault-injected apps are generated one at a time and sent for differential analysis. If no evidence of malicious behavior is found in the differential analysis, the fault injection process is invoked again to generate a different faulty app, and so on.

Assume that $\mathsf{CoI} = \{c_1, c_2, \ldots, c_n\}$ and that for each $c_i \in \mathsf{CoI}$ there is a set of FIOs $\mathcal{F}_i = \{\psi_{i_1}^{c_i}, \psi_{i_2}^{c_i}, \ldots, \psi_{i_{m_i}}^{c_i}\}$ that can be applied to c_i. Recall that FIOs can be quite specific and, therefore, not all FIOs are applicable to all components. All possible fault-injected apps can be generated by a naïve strategy that applies each FIO to each component one at a time:

$$\psi_{i_1}^{c_1}(\mathcal{P}), \ldots, \psi_{i_{m_1}}^{c_1}(\mathcal{P}), \ldots, \psi_{i_1}^{c_n}(\mathcal{P}), \ldots, \psi_{i_{m_n}}^{c_n}(\mathcal{P}) \tag{5.16}$$

Thus, there are $\sum_{j=1}^{n} m_j$ possible fault-injected apps, one for each possible component-FIO pair.

All FIOs in Alterdroid are indistinguishable. This allows for a more efficient fault injection process based on the fact that the composition of indistinguishable FIOs is an indistinguishable FIO. Consequently, if the same FIO is applied to multiple components and there is hidden functionality in just one of them, the resulting app will behave exactly as if just the malicious component would have been fault injected. The resulting fault injection process is as follows:

1. For each FIO ψ_j, generate $\mathcal{P}_j'$ by applying it to all $c_i \in \mathsf{CoI}$

$$\mathcal{P}_j' = \Psi_j(\mathcal{P}) = \psi_{i_j}^{c_1} \circ \psi_{i_j}^{c_2} \circ \cdots \circ \psi_{i_j}^{c_n}(\mathcal{P}) \tag{5.17}$$

   where $\psi_j^{c_i}$ is the void operator if Ψ_j is not applicable to c_i. The resulting P_j' is sent for differential analysis with respect to the original $\mathcal{P}$.

2. If there is one $\mathcal{P}_j'$ such that the differential analysis spots malicious behavior, the component responsible for it can be identified by searching over all $c_i \in \mathsf{CoI}$ with just the corresponding FIO Ψ_j. This process can be done in logarithmic

time by ordering all components and recursively applying Ψ_j to half of them, rather than in linear time by just applying Ψ_j to each $c_i \in \mathsf{Col}$ in turn.

The overall process, which is entwined with the differential analysis stage discussed later, is summarized in the algorithm shown in Figure 5.3. Note that in this description the process stops when just one malicious component is identified. Extending the algorithm to search for all of them is straightforward.

5.3.3 Applying Differential Analysis

Differential analysis between a candidate fault-injected app and the original app is carried out by following the model described in Sections 5.2.2 and 5.2.3. The process comprises the following steps:

1. Generate an appropriate usage pattern $\mathbf{u}$ and context $\mathbf{g}$ [Rastogi et al., 2013a; Zheng et al., 2012] to feed both apps and extract their behavioral signatures, $\beta[\mathcal{P}(\mathbf{u}|\mathbf{g})]$ and $\beta[\mathcal{P}'(\mathbf{u}|\mathbf{g})]$. Both the original and the fault-injected app are tested under the same conditions and using the same input. Note that this assumes that the execution of an app is completely deterministic.

2. Generate the differential signature $\Delta(\beta[\mathcal{P}(\mathbf{u}|\mathbf{g})], \beta[\mathcal{P}'(\mathbf{u}|\mathbf{g})])$ from the behavioral signatures obtained above.

3. Apply sequentially all rules $\mathbf{R}_i$ over $\Delta(\beta[\mathcal{P}(\mathbf{u}|\mathbf{g})], \beta[\mathcal{P}'(\mathbf{u}|\mathbf{g})])$ and return those that match.

The process is summarized in the algorithm given in Figure 5.4.

Algorithm 2. Fault Injection and Malicious Search

Input:
App: $\mathcal{P}$
Col $= \{c_1, c_2, \ldots, c_n\}$
Set of FIOs: $\mathcal{F} = \{\psi_1, \psi_2, \cdots, \psi_m\}$

Output:
List of all malicious components

Algorithm:

```
1  maliciousComp ← null
2  For each FIO ψ_j do
3       P'_j ← P
4       For each c_i ∈ Col do
5            if ψ_j is applicable to c_i then
6                 P'_j = ψ_j^{c_i}(P)
7            end-if
8       end-for
9       if DiffAnalysis(P, P', ψ_j) ≠ ∅ then
10           maliciousComp ← SearchComponent(ψ_j, P, Col, 1, n)
11      end-if
12 end-for
13 return maliciousComp

Function SearchComponent(ψ_j, P, Col, min, max)
1  P'_j ← P
2  For i = min to max do
3       if ψ_j is applicable to c_i then
4            P'_j = ψ_j^{c_i}(P)
5       end-if
6  end-for
7  if DiffAnalysis(P, P', ψ_j) ≠ ∅ then
8       if min = max then
9            return c_min
10 else
11           SearchComponent(ψ_j, P, Col, min, (max − min)/2)
12           SearchComponent(ψ_j, P, Col, (max − min)/2), max)
13 end-if
```

Figure 5.3: Algorithm for injecting faults and searching for malicious components after differential analysis.

Algorithm 3. Differential Analysis

Input:
Apps: $\mathcal{P}$ and $\mathcal{P}'$
FIO ψ
Set of rules: $\mathcal{R} = \{\mathbf{R}_1, \mathbf{R}_2, \ldots, \mathbf{R}_p\}$
Output:
Matching rules
Algorithm:

```
1  (u, g) ← GenUsagePatterns(P)
2  β ← GenBehavioralSig(P, u, g)
3  β' ← GenBehavioralSig(P', u, g)
4  Δ(β, β') ← ComputeDiffSig(β, β')
5  matchingRules ← ∅
6  For each R_i ∈ R do
7      if match(R_i, ψ, Δ(β, β')) then
8          matchingRules ← matchingRules ∪ {R_i}
9      end-if
10 end-for
11 return matchingRules
```

Figure 5.4: Algorithm DiffAnalysis for generating differential signatures and identifying matching rules.

5.3.4 Implementation

Alterdroid is implemented over our Maldroid Lab described in Chapter 3. App components are extracted and later on (after fault injection) repackaged using our static analysis component. We then generate common sequences of events and execute each app dynamically. In order to generate behavioral signatures, Alterdroid monitors the execution of the following different activities: *crypto*, *netopen*, *netread*, *netwrite*, *fileopen*, *fileread*, *filewrite*, *sms*, *call*, *leak*, and *dexload*.

Our prototype allows performing analysis tasks in parallel. We presently limit our implementation to a small number of CoI models, FIO operators, and differential matching operators. Nonetheless, our architecture allows security experts to further

extend this and configure their own operators based on their experience.

5.3.4.1 CoI Models

Alterdroid currently supports the following models for identifying CoIs:

- `EXEFileMatch`. This model analyzes components of type Dalvik Executable Format (`DEXFileMatch`), Application Package file format (`APKFileMatch`), and Executable and Linkable Format (`ELFFileMatch`), i.e., $\tau(c) = \langle DEX, APK, ELF \rangle$. The model defined for these components is based on the magic number defined in the header of the file.

- `ImgFileMatch`. This model analyzes components of type picture, such as PNG, JPG, or GIF images, i.e., $\tau(c) = \langle PNG, \cdots, JPG \rangle$. This model is based on the magic number defined in the file header, similarly to the model above.

- `EncryptedOrCompressedMatch`. This model matches any file whose entropy, measured at the byte level, exceeds a given threshold. In such a case, the file is considered to contain random or encrypted information and, therefore, is selected for fault analysis. We set the current threshold to 3.9. Such value was chosen after measuring the entropy of several files before and after being encrypted with DES.

- `ExtensionMismatch`. This model identifies files such that their magic numbers do not match the file extension. For instance, we found several *APK* files with *DB* extension and several encrypted files with *JPG* extension. We currently support two submodels: `ImgFileExtensionMismatch` and `APKFileExtensionMismatch`.

FIO	Type	Targeted CoIs	ind
GenericFMutation	Any file	ImgExtensionMismatch	✓
		EncryptedOrCompressed	–
		APKFExtensionMismatch	✓
ImgFileChange	Any image	ImgFileMatch	✓
ScriptFileChange	Non-compiled program	TextScriptMatch	×
APKFileChange	Android app	APKFileMatch	×
DEXFileChange	Dalvik executable	DEXFileMatch	×
ELFFileChange	Executable and linkable	ELFFileMatch	×

Table 5.1: FIOs implemented in Alterdroid's current version and their corresponding CoIs.

- `TextScriptMatch`. This model analyzes components that match any ASCII text executable file, i.e., $\tau(c) = \mathit{Script}$. This model is also based on the magic number defined in the file header.

All CoIs described above are implemented in Python. The set can be easily extended to incorporate additional models by simply adding the corresponding Python module.

5.3.4.2 Fault Injection Operators

FIOs in Alterdroid are strongly typed. This avoids syntactic or unexpected errors during the execution of the modified app. For instance, if a generic FIO modifies randomly chosen bits of a JPEG without considering the file structure, it may end up with a malformed picture that could cause the app to crash during execution. We currently support the next list of FIOs (see also Table 5.1):

- `ImgFileChange`. This FIO changes a number of pixels of image file components. The FIO type matches components of type `ImgFileMatch`. This is an

indistinguishable FIO due to the nature of the changes and the type of component. Thus, although the image resulting from the injection will be different, this change should not alter the app execution flow.

- `EXEFileChange`. This FIO replaces the file with a well-formed *APK*, *DEX* or *ELF* file that effectively does nothing, equivalent to a NOP (no-operation) injection. This change should cause a different behavior in the resulting differential signature as the former *EXE* file has been replaced. Thus, this FIO is distinguishable.

- `ScriptFileChange`. This FIO replaces the file with a valid NOP script. It only matches components of type `ScriptFileChange`. This FIO is also distinguishable.

- `GenericFileMutation`. It randomly changes several bytes of a file. This FIO is applied when there is no information about the file type and its structure, e.g., when injecting faults to encrypted files (`EncryptedOrCompressedMatch`) or when the file extension does not match its magic number `ExtensionMismatch`. This FIO might be distinguishable or indistinguishable, depending on the file type.

As in the case of CoI models, FIOs are implemented in Python and provided with Alterdroid's current version. Again, the set can be easily extended with additional FIOs by adding the corresponding Python module.

Name	Contains	Rule
$\mathbf{R_{NAC}}$	Net. Activity Component	$\mathsf{ind}(\psi^c) \wedge \exists\ i_1 : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i_1=netopen}\big)$ $\vee \exists\ i_2 : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i_1=netread}\big)$ $\vee \exists\ i_3 : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i_2=netwrite}\big)$
$\mathbf{R_{FAC}}$	File Activity Component	$\mathsf{ind}(\psi^c) \wedge \exists\ i_1 : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i_1=fileopen}\big)$ $\vee \exists\ i_2 : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i_1=fileread}\big)$ $\vee \exists\ i_3 : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i_2=filewrite}\big)$
$\mathbf{R_{DLC}}$	Data Leakage Component	$\mathsf{ind}(\psi^c) \wedge \exists\ i\ : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i=leak}\big)$
$\mathbf{R_{SAC}}$	SMS Activity Component	$\mathsf{ind}(\psi^c) \wedge \exists\ i\ : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i=sms}\big)$
$\mathbf{R_{PAC}}$	Payload Activity Component	$\mathsf{ind}(\psi^c) \wedge \exists\ i\ : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i=dexload}\big)$
$\mathbf{R_{UPC}}$	Update Payload Component	$\mathsf{ind}(\psi^c) \wedge \exists\ i_1 : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i_1=netread}\big)$ $\wedge \exists\ i\ : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i=dexload}\big)$
$\mathbf{R_{CAC}}$	Crypto Activity Component	$\mathsf{ind}(\psi^c) \wedge \exists\ i\ : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i=crypto}\big)$
$\mathbf{R_{CPC}}$	Crypto Payload Component	$\mathsf{ind}(\psi^c) \wedge \exists\ i_1 : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i_1=crypto}\big)$ $\wedge \exists\ i\ : \mathsf{contains}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \mathsf{Del}_{i=dexload}\big)$
$\mathbf{R_{HFC}}$	Hidden Functionality Component	$\mathsf{ind}(\psi^c) \wedge \neg\mathsf{equal}\big(\Delta(\beta[\mathcal{P}], \beta[\psi^c(P)]), \emptyset\big)$

Table 5.2: Basic indistinguishable differential rules implemented in Alterdroid.

5.3.4.3 Differential Rules

The basic set of differential rules incorporated in Alterdroid comprises the 9 rules shown in Table 5.2. They all apply to indistinguishable FIOs and cover the most common examples of obfuscated functionalities: network activity, file activity, data leakage, SMS activity, hidden payloads, update attacks, cryptographic activity, cryptographic payloads, and generic hidden functionality.

To reduce the complexity of the search space, all basic rules apply to indistinguishable FIOs. However, for the sake of completeness our implementation incorporates several distinguishable FIOs, and new rules can be further added to match them. For instance, given an app that incorporates a DEX program used to enhance photos

taken from the camera, we can use a rule to check whether this CoI actually does just that or not.

Thus, if after applying a FIO over this component the differential signature shows, for instance, changes in network activity, we may suspect that the CoI contained other functionality piggybacked on the DEX.

Formally, given `DEXFileMach` $\in$ `CoIs` and its corresponding distinguishable FIO (i.e., `DEXFileChange`), the following rule captures this intuition:

$$\begin{aligned} &\mathbf{R_{DEX}} \ : \ \textit{dex} \in \mathcal{P} \text{ contains NET activity } \Longleftrightarrow \\ &\neg \mathsf{ind}(\Psi^{\textit{dex}}) \ \wedge \ \exists \ i \ : \ \mathsf{contains}\big(\Delta(\sigma[\mathcal{P}], \sigma[\Psi^{\textit{dex}}(\mathcal{P})]), \mathsf{Del}_{i=net}\big) \end{aligned} \tag{5.18}$$

Note that we limit our implementation to a small number of indistinguishable FIOs and matching rules. Nonetheless, our architecture allows security experts to further extend this and configure their own FIOs and rules based on their experience.

5.4 Evaluation

We next report a number of experimental results obtained with our prototype implementation of Alterdroid. These results illustrate how our system can be used by market operators and security analysts to facilitate the analysis of complex obfuscated mobile malware. We first present the results of testing Alterdroid against two datasets of smartphone malware samples found in the wild, including a performance analysis of the entire differential fault analysis process. We finally discuss in more detail three representative case studies.

5.4.1 Analytical Results

We tested Alterdroid against a dataset composed of around 6K apps retrieved from Aptoide (AP) alternative market and VirusShare (VS) repository (see Chapter 3—Section 3.5.2). Every app was executed over a time span of 120 seconds—current malware is generally quite eager to run their payloads promptly [Suarez-Tangil et al., 2014a], so this time suffices to activate most malicious payloads. Table 5.3 provides a summary of the obtained experimental results, including the average time required for analyzing one app (this includes the time for extracting CoIs and injecting faults into them.

When analyzing the distribution of CoIs throughout the apps in our datasets, we observed that some apps have a fairly large amount of CoIs (see Figure 5.5). For instance, we can find some apps with over 5K images (`ImgFileMatch`). Conversely, we could find many apps with fewer CoIs. On average, our experiments show that there are about 146 and 284 CoIs per app in VS and AP respectively, as shown in Table 5.3. Note that the number of CoIs from AP is twice the number of CoIs from VS. In any case, the amount of potentially malicious components is significant and the time required to analyze each of them manually is shown affordable.

Finally, our results report a number of apps matching against the rules implemented in our prototype. For instance, we could identify 220 apps reporting components containing SMS functionality ($\mathbf{R_{SAC}}$) from all 2.9K samples in VirusShare. Conversely, we could not find any $\mathbf{R_{SAC}}$ rule in Aptoide (see Table 5.3). One alarming result is that we found a significant number of apps, i.e.: 669, reporting components containing data leakage functionality ($\mathbf{R_{DLC}}$) in Aptoide.

One interesting aspect of Alterdroid is that it can inject all selected FIOs at once.

		VirusShare (VS)	Aptoide (AP)
Sum.	**No. Apps**	2 913	2 994
	Avg. No. Cols	145.6	284.4
	Avg. No. FIOs	138.3	273.5
Cols	**ImageFileMatch**	3279	5215
	EncryptedOrCompressed	16 687	35 293
	ImageExtensionMismatch	5 771	5 246
	DEXFileMatch	2 827	2 995
	APKFileMatch	1 087	58
	APKExtensionMismatch	517	39
FIOs	**ImageFile**	397 248	813 754
	GenericMutationFile	5 714	5 237
Rules	**No. R_{FAC}**	2 802	2 962
	No. R_{NAC}	2 773	2 929
	No. R_{DLC}	1 971	669
	No. R_{SAC}	220	0
–	**Avg. Overhead**	584.51 s.	666.67 s.

Table 5.3: Analysis of the VS and AP datasets. The number of Cols and FIOs is given on average per app. The number of matches (NAC and DLC) is given in absolute value, and the overhead is on average per app.

Furthermore, Alterdroid allows performing several analyses concurrently. In fact, our current experimental setup allows the execution of 15 instances of Android in parallel. Thus, this simple optimization strategy reduces the average execution time per app at 32.62 and 44.44 seconds for VS and AP, respectively.

One challenge we faced when analyzing apps from Aptoide is identifying whether some behaviors were malicious or not. Many legitimate apps are not fully malicious but carry out activities that may constitute a privacy risk for some users. During our analysis, most such suspicious behaviors were related with accessing local data and exfiltrating it over the network. We did not analyze in detail whether this was an intrinsic behavior of the app caused by the fault-injection process, for example because the app contained an integrity check. Nonetheless, this indicates that the

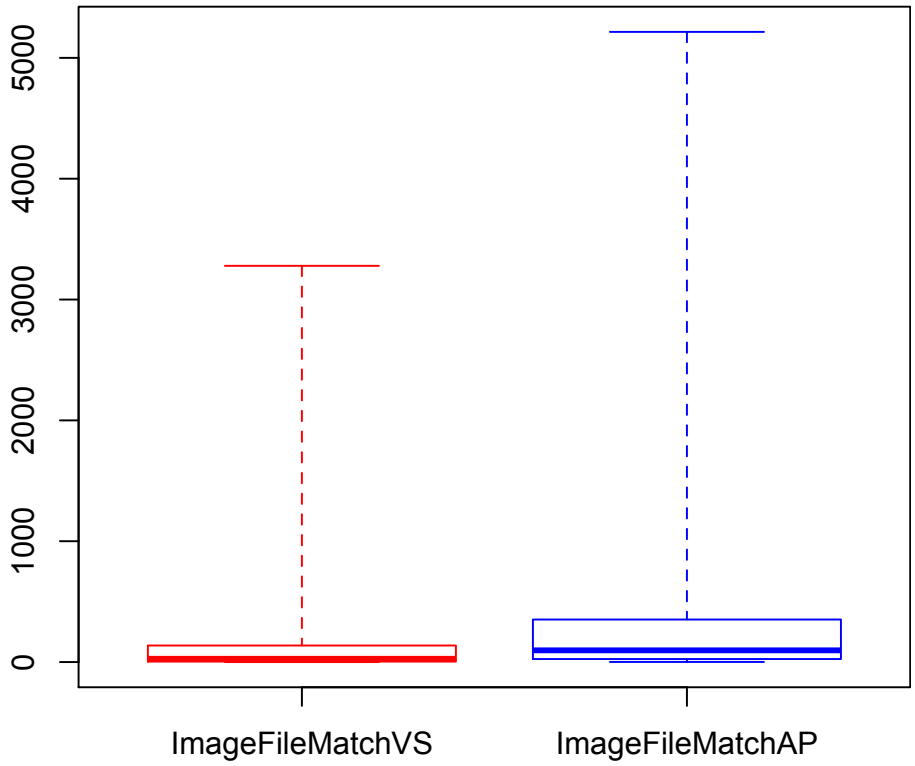


Figure 5.5: Distribution of the number of *ImageFileMatch* in the VirusShare (VS) and Aptoide (AP) datasets.

app was behaving suspiciously and therefore it is worth analyzing.

5.4.2 Performance

The time taken by the entire differential analysis process depends on the number of different fault-injected apps to be explored, the time required to generate each of them, and the time taken by the differential analysis over each one:

$$t = n_{\mathsf{faultApps}} \cdot t_{\mathsf{genFaultApp}} \cdot t_{\mathsf{diffAnalysis}} \tag{5.19}$$

As for the first term, if $|\mathsf{Cols}| = n$ and there are m FIOs, the fault injection

algorithm shown in Figure 5.3 generates $O(m + \log n)$ different fault-injected apps to be analyzed. Each one of those apps has been injected with at most n faults, one per component. The time $t_{\mathsf{genFaultApp}}$ required to inject one fault depends on the specific FIO, although most of them run in constant time or are linear in the size of the component to be fault-injected. Finally, differential analysis requires:

- Executing the two apps. In Alterdroid this is done by a component which admits as input the time t_{exec} during which the app will be executed.

- Obtaining the differential signature, which reduces to computing an edit distance between the two activity signatures. If these signatures have lengths h_1 and h_2, then this process takes $O(h_1 \cdot h_2)$ steps.

- Pattern-matching the differential signature with the rule-set, which takes $O(|\mathcal{R}|)$.

Apart from t_{exec}, the two most critical parameters affecting the total analysis time are n and m, as defined above (i.e., number of CoIs and FIOs, respectively). Fig. 5.6 shows the average execution time of the SearchComponent identification algorithm at the core of Alterdroid for different values of n, m, and t_{exec}. For example, the analysis of an app containing 100 CoIs for which 10 FIOs are applicable, and executing each fault-injected app 120 s, will require around 5 minutes. The time increases to 2.5 hours and 4.5 hours if the app contains 1K or 10K CoIs, respectively. If we decrease the dynamic execution time of each app to 60 s, these figures reduce to 2.7 minutes, 1.3 hours, and 2.9 hours, respectively.

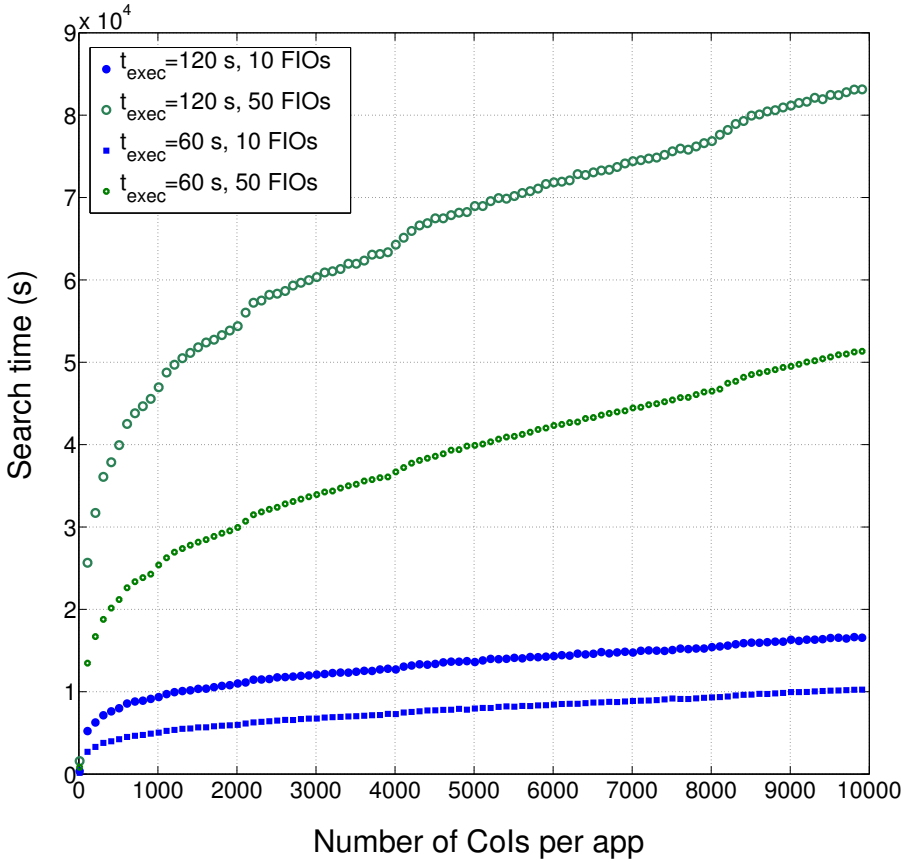


Figure 5.6: Average execution time of the SearchComponent algorithm for different number of FIOs and dynamic analysis time.

5.4.3 Case Studies

We next illustrate how Alterdroid can be used by market operators and security analysts to facilitate the analysis of complex obfuscated mobile malware. We present three case studies of malicious apps found in Android markets: *GingerBread*, *DroidKungFu*, and *AnserverBot*. These three samples constitute representative cases as they incorporate obfuscation techniques of various degrees of sophistication, as well as some malicious features common in malware for smart devices (see Chapter 2) such as aggressive privilege escalation exploits, C&C-like functions and information leakage. Figure 5.7 summarizes the behavior of the malware that will be discussed

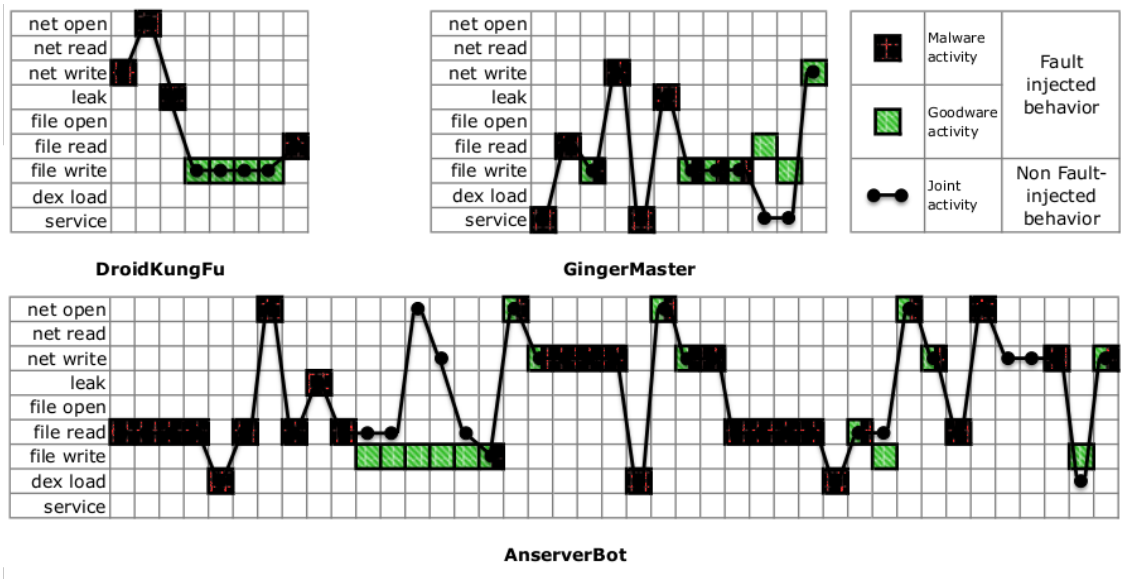


Figure 5.7: Malware's activity during a time span of 120 seconds. The x-axis represent the sequence of activities observed during the execution.

throughout this section. Subsequently, we evaluate the performance of our approach over a number of malware samples found in the wild [Zhou and Jiang, 2012].

5.4.3.1 DroidKungFu

DroidKungFu (DKF) is one of the major Android malware outbreaks. DKF's main goal is to collect a variety of information on the infected device, including the IMEI number, phone model, as well as the Android OS version.

DKF is mostly distributed through open or alternative markets through repackaging, i.e., by piggybacking the malicious payload into a variety of legitimate applications. Apps infected with DKF are distributed together with a root exploit hidden within the app's assets, namely Rage Against the Cage (RAC). In order to hinder static analysis, this encrypted payload is only decrypted at runtime.

In this case study, we analyze one DKF variant by first extracting its components

of interest and further applying fault injection and differential analysis over them. We observed that the sample contained about 170 resource files, including PNG (153 files), MP3 (6 files), XML (2 files), DEX (1 file) and RSA key file, among others. All these assets are, in principle, suspected of containing obfuscated functionality. We note here that applying stand-alone static detection techniques would not be enough to identify malicious payloads without requiring human-driven inspections. This is due to the way that DKF obfuscates its core components. Specifically, each variant uses a different encryption key hidden throughout the code. Even when we attempt to apply stand-alone dynamic analysis, we observe that this technique only gives a rough notion of the holistic behavior of the app. In fact, the behavior introduced by DKF is strongly entwined with the original code of the repackaged app, in such a way that some of its key activities, such as for instance network connections, might be easily seen as normal.

The above-mentioned variant of DKF was fed to Alterdroid. It first identified a number of components of interest, being in all the cases assets associated with the app. Various faults were then injected into such components, and the resulting app was executed and compared with the original one. Figure 5.7 (DroidKungFu) graphically shows the differential behavior reported by Alterdroid when analyzing such fault-injected app. Activities launched by the original piggybacked app correspond to the full plot, while the behavior after fault injection is given just by the green (legitimate app) and red spots (DKF). In this particular case, a text file pertaining to the assets was randomly modified. This file was later identified as the component containing the RAC exploit. Our analysis shows that disabling the access to such a functionality stops the malware from: establishing a network connection (*netopen*, *netwrite*), leaking some information through it (*leak*), and later performing some

Input-Output (I/O) operations (*fileread*). These findings agree with previous reports about DKF, including those undertaken by Jiang and Zhou [Zhou and Jiang, 2012].

5.4.3.2 AnserverBot

Our second case study deals with *AnserverBot* (ASB), a specimen similar to the first versions of DKF in terms of sophistication and distribution strategy [Suarez-Tangil et al., 2014c]. However, ASB introduces an update component that allows it to retrieve at runtime secondary payloads and the latest C&C URLs from public blogs. Additionally, it also incorporates advanced anti-analysis methods to avoid detection. On the one hand, it introduces an integrity component to check if the app has been modified. On the other hand, it piggybacks the main payload in native runnable code. Furthermore, ASB obfuscates its internal classes and methods, and partitions the main payload in two different parts: while one of them will be installed, the other one is dynamically loaded without actually being installed. More specifically, ASB hides one of these components into the assets folder under any of the following names: *anservera.db* or *anserverb.db*. Furthermore, ASB inserts a new component named *com.sec.android.provider.drm* that executes a root exploit known as *Asroot* [Grace et al., 2012b].

As in the case of DKF, we observed that all ASB samples contain a non-negligible amount of candidate components to be analyzed. The specimen we deal with in this case study contained about 78 resource files, including 54 image files, one database, one DEX file, and a ZIP file, to name a few. After a few iterations of the fault injection process Alterdroid succeeds in positively identifying the actual payload within the DB file, as well as the behavior related to such component. More precisely, this

CoI is triggered after observing a mismatch between the magic number of the file (APK) and the actual extension of the data base (DB). In fact, when a fault is injected over the database, the ASB's integrity check naturally aborts its execution and produces a result similar to that expected from the original app. Figure 5.7 (AnserverBot) graphically shows the exhibited differential behavior. As observed, ASB first establishes a network connection (*netopen* and *netwrite*) after loading the main payload (*fileread* operations followed by *dexload*). After that, it keeps on reading data that is finally leaked out. Interestingly, the legitimate application uses the network as well, although it does not leak any personal information.

5.4.3.3 GingerMaster

GingerMaster (GM) was the first known Android malware to use root exploits for privilege escalation on Android 2.3. GM's main goal is to exfiltrate private information such as the device ID (IMEI, MSI, etc.) or the contact list stored in the phone. GM is generally repackaged with a root exploit known as *GingerBreak* [Grace et al., 2012b], which is stored as a PNG and a JPG asset file. Right after infecting the device, GM connects to the C&C server and fetches new payloads.

We analyzed a GM sample containing around 61 asset resources, 30 of which were pictures in different formats. From those 30 pictures, Alterdroid identified 4 as strongly suspicious. (Actually, a detailed analysis shows that they are malformed PNGs and that they also contain several ASCII text scripts.) Alterdroid was also able to identify that such malformed images files play a key role in triggering the payloads piggybacked into the legitimate app, including the ASCII text scripts.

Figure 5.7 (GingerMaster) shows the differential behavior obtained when one of

such images is fault injected. Our analysis shows that GM starts the execution of a *service* that performs some IO operations (*file-read* and *file-write*) before finally leaking private information through the network (*net-write* and *leak*). Again, even when the malicious components are hidden, Alterdroid proved to be able to discriminate them and facilitate the identification of the underlying malicious behavior.

5.5 Conclusions

Today's mobile security requires new approaches to protect users' devices as traditional detection techniques are overwhelmed by the sophistication and obfuscation of current mobile malware [Leavitt, 2013]. Furthermore, the current panorama and trends suggest that automated malware detection and analysis is a major requirement for apps review.

Differential fault analysis in the way implemented by Alterdroid is a powerful and novel dynamic analysis technique that can identify potentially malicious components hidden within an app package. Additionally, empowering dynamic analysis with a fault injection approach can be used to differentiate the "gray" behavior from the legitimate when analyzing *grayware*. This is a good complement to static analysis tools, more focused on inspecting code components but which could well miss pieces of code hidden in data objects or just obfuscated.

Alterdroid is thought as a general purpose framework with a very versatile architecture that can be extended in a number of ways. In this chapter, we have described this architecture together with a formal notion of differential fault analysis. Additionally, we present an open-source engineered version Alterdroid striving on offering an automated tool for malware analysis. Furthermore, based on our experimental re-

sults, we reduced from 6K apps to 2.6K apps suspicious of containing data leakage functionality. In addition, performance figures are given and discussed, showing the feasibility of such a novel approach to differential analysis. Even though Alterdroid is presently a perfectly functional proof of concept, its open architecture and available open sources can make it the basis for further research work and production/professional software.

Although current malware is relatively naïve, more sophisticated obfuscation techniques—particularly in code—are starting to materialize. Cryptography is one recurrent technique used by malware developers. Nonetheless, we believe that malware could be already using other advanced techniques for hiding their components such as, for instance, Steganography. This technique would allow them to conceal their malicious components within other objects of the code. This is specially critical when these components are hidden within distinguishable components.

IV

Cloud-based Analysis

6

Power-aware Anomaly Detection in Smartphones

6.1 Introduction

Many security issues can be essentially reduced to the problem of separating malicious from non-malicious activities. Such a reformulation has turned out to be valuable for many classic computer security problems, including detecting network intrusions, filtering out spam messages, or identifying fraudulent transactions. But, in general, defining in a precise and computationally useful way what is harmless or what is offensive is often too complex. To overcome these difficulties, many solutions to such problems have traditionally adopted a machine learning approach, notably through the use of classifiers to automatically derive models of good and/or bad behavior that could be later used to identify the occurrence of malicious activities.

Anomaly-based detection strategies have proven particularly suitable for scenarios where the main goal is to separate "self" (i.e., normal, presumably harmless behavior) from "non-self" (i.e., anomalous and, therefore, potentially hostile activities). In

this setting, one often uses a dataset of self instances to obtain a model of normal behavior. In detection mode, each sample that does not fit the model is labelled as anomalous. This notion has been thoroughly explored over the last two decades and applied to multiple domains in the security arena [Chandola et al., 2009; Estévez-Tapiador et al., 2004; Garcia-Teodoro et al., 2009].

More recently, many security problems related to smartphone platforms have been approached with anomaly-based schemes (see, e.g., [Burguera et al., 2011; Dini et al., 2012; Feizollah et al., 2014; Rosen et al., 2013; Shabtai et al., 2012]). One illustrative example is found in the field of continuous—or implicit—authentication through behavioral biometrics [De Luca et al., 2012; Jakobsson et al., 2009; Shi et al., 2011]. The key idea here is to equip the device with the capability of continuously authenticate the user by monitoring a number of behavioral features, such as for example the gait—measured through the built-in accelerometer and gyroscope—, the keystroke dynamics, the usage patterns of apps, etc. These schemes rely on a model learned from user behaviors to identify anomalies that, for example, could mean that the device is mislaid, in which case it should lock itself and request a password.

Proposals for detecting malware in smartphones have also made extensive use of anomaly detection approaches. Most schemes are built upon the hypothesis that malicious apps somehow behave differently from goodware. The common practice consists of monitoring a number of features for non-malicious apps, such as for example the amount of CPU used, network traffic generated, system/API calls made, permissions requested, etc. These traces are then used to train models of normality that, again, can be used to spot suspicious behavior. Modeling app behavior in this way is particularly useful in two scenarios. The first one is related to the problem of

repackaged apps, which constitutes one of the most common distribution strategies for smartphone malware. In this case, the malicious payload is piggybacked into a popular app and distributed through alternative markets. Detecting repackaged apps is a challenging problem, in particular when the payload is obfuscated or dynamically retrieved at runtime. The second problem is thwarting the so-called grayware, i.e., apps that are not fully malicious but that entail security and/or privacy risks of which the user may not be fully aware. For instance, an increasing number of apps access user-sensitive information such as locations frequently visited, contacts, etc., and send it out of the phone for obscure purposes [Kranz et al., 2013]. As users find it difficult to define their privacy preferences in a precise way, automatic methods to tell apart good from bad activities constitute a promising approach.

Essentially all machine learning-based anomaly detection solutions can be broken down into the following functional blocks:

- *Data acquisition*. Activity traces are required both for (re-)training the model of normality and in detection mode. The nature of the data collected varies across applications and may include events such as system calls, network activities, user-generated inputs, etc.

- *Feature extraction*. Machine learning algorithms require data to be expressed in particular formats, commonly in the form of feature vectors. A number of features are extracted from the acquired activity traces during a preprocessing stage. The complexity of such preprocessing depends on the problem and ranges from computationally straightforward procedures (e.g., obtaining simple statistics from the data) to more resource intensive transformations.

- *Training*. A representative set of feature vectors is used to train a model that

captures the underlying notion of normality. This process may be done offline, in which case periodic re-training is often necessary in order to adapt the model to drifts in behavioral patterns, or else constantly as new data arrives.

- *Detection*. Once a behavioral model is available, it is used along with a similarity function to obtain an anomaly score for each observed feature vector. This process is often carried out in real time and requires constant data acquisition and feature extraction.

All the functions described above can be quite demanding—particularly if they must operate constantly—and it is debatable whether they can be afforded in energy-constrained devices with limited computational capabilities. As a consequence, a number of recent works (see, e.g., [Barbera et al., 2013; Portokalidis et al., 2010; Zonouz et al., 2013]) have suggested externalizing some of these tasks to dedicated servers in the cloud or to other mobile devices nearby [Yu et al., 2013]. However, these proposals do not provide a detailed analysis of the cost in terms of energy consumption.

Although off-loading computation seems intuitively advantageous, such a strategy has an implicit trade-off between the energy savings resulting from not performing on-platform computations and the costs involved in data exchanges over the network. Intermediate strategies are also possible, such as for example off-loading the training stage only and performing detection locally, or externalizing everything but the data acquisition and preprocessing stages. Additionally, each plausible placement strategy has consequences in aspects other than energy consumption. For example, off-loaded detection may result in delays in detecting anomalous events, or even malfunctions if network connectivity is unavailable.

Intuition suggests that intensive monitoring is prohibitive for platforms such as the current generation of smartphones [Rachuri et al., 2014]. However, the energy consumption trade-offs among the various on-platform and externalized computation strategies are unclear. Several works (e.g., [Kumar and Lu, 2010; Namboodiri and Ghose, 2012; Tandel and Venkitachalam, 2013]) have addressed the issue of deciding whether *to cloud* is a better option than *not to cloud* for mobile systems. For example, in [Namboodiri and Ghose, 2012] it is shown that determining an energy efficient strategy is a complex task and require a fine characterization of the impact of several parameters, including the type of device and the application domain. Their approach focuses on three rather generic applications: word processing, multimedia, and gaming for both laptops and mobile devices. Authors conclude that "cloud-based applications consume more energy than non-cloud ones" when using platforms such as mobile devices. In contrast, other works such as [Tandel and Venkitachalam, 2013] and [Kumar and Lu, 2010] show that offloading is generally profitable energy-wise, particularly for intensive computation tasks that require relatively small amount of communications.

In this chapter, we address the problem discussed above and assess the energy-consumption trade-offs among different strategies for off-loading, or not, functional tasks in machine learning based anomaly detection systems. Our analysis is motivated by, and hence strongly biased towards, security applications of anomaly detectors, such as for example malware detection or behavioral authentication. Nevertheless, the majority of our experimental setting, results and conclusions are general and may be of interest to other domains where smartphone-based anomaly detectors are used (e.g., health monitoring applications [Kranz et al., 2013]).

In summary, our results confirm the intuition that externalized computation is,

by far, the best option energy-wise. However, one rather surprising finding is that it is several orders of magnitude cheaper than on-platform computations, which suggests that networking is much more optimized than computation in such platforms. Furthermore, we have noticed substantial differences among the machine learning algorithms tested. Since some of them appear not to scale well for large feature vectors and/or datasets, developers should make careful choices when opting for one algorithm or another. In addition, anomaly detectors are found to consume considerably more energy than popular apps such as games or online social networks, which motivates the need for more lightweight machine learning algorithms.

The rest of the chapter is organized as follows. Section 6.2 describes the experimental setting used in our work, including the platform used, the anomaly detectors tested and the experiments carried out. Empirical results are discussed in Section 6.3, and energy-consumption linear models are numerically derived for each separate function. Such models are used in Section 6.4 to analyze various off-loading strategies and provide a comparative discussion. In Section 6.5 we illustrate the main findings discussed throughout the chapter using an anomaly-based detector of repackaged malware. Section 6.6 concludes the chapter by summarizing our contributions and main conclusions.

6.2 Experimental Setting

In this section, we describe the experimental framework used for evaluating energy consumption in Android devices, including the machine learning algorithms evaluated, the tests carried out, and the tools and operational procedures used to measure power.

6.2.1 Machine Learning Algorithms

We have tested three machine learning algorithms that can be used as anomaly detectors. Our choosing of these particular schemes is motivated by the different computational approaches followed by each one of them, and also because they are representative of broad classes of machine learning strategies: decision trees [Quinlan, 1986], clustering [Fisher, 1987], and probabilistic approaches [Hastie et al., 2005]. For completeness, we next provide an overview of each algorithm's working principles.

- *J48* is a Java implementation of the classic C4.5 algorithm [Hastie et al., 2005]. The procedure builds a decision tree from a labelled training dataset using information gain (entropy) as a criterion to choose attributes. The algorithm starts with an empty tree and progressively grows nodes by choosing those attributes that most effectively split the dataset into subsets where one class dominates. This procedure is recursively repeated until reaching nodes where all instances belong to the same class [Hastie et al., 2005].

  The resulting tree can be used as a classifier that outputs the class of future observations based on their attributes. The binary setting (i.e., two classes: normal and anomalous) is commonly used in anomaly detection problems, although it is perfectly possible to train a classifier with more a complex class structure.

- *K-means* is a clustering algorithm that groups data into k clusters and returns the geometric centroid of each one of them. Given a dataset composed of feature vectors $\mathcal{D} = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$, the algorithm searches for a partition of $\mathcal{D}$

into k clusters $\{s_1, \ldots, s_k\}$ such that the within-cluster sum of squares

$$\sum_{i=1}^{k} \sum_{\mathbf{x_j} \in s_i} \| \mathbf{x}_j - \mu_i \|^2 \tag{6.1}$$

is minimized, where μ_i is the geometric mean of the vectors in s_i.

When used in a supervised training setting, each centroid μ_i receives a class label derived from the labels of the samples associated with the corresponding cluster. Labelled centroids can be then used, together with a nearest neighbor classifier, to determine the class of an observation by simply assigning it to a cluster according to some distance. Clustering algorithms have been extensively used in anomaly detection, particularly in one-class settings where only normal training instances are available. In such cases, a sample is often labelled as anomalous if its sufficiently far away from its nearest centroid.

- *OCNB (One Class Naïve Bayes)* [Hastie et al., 2005] is a supervised learning algorithm that has been successfully used in a wide range of applications. OCNB is often a very attractive solution because of its simplicity, efficiency and excellent performance. It uses the Bayes rule to estimate the probability that an instance $x = (x_1, \ldots, x_m)$ belongs to class y as

$$P(y|x) = \frac{P(y)}{P(x)} P(x|y) = \frac{P(y)}{P(x)} \prod_{i=1}^{m} P(x_i|y) \tag{6.2}$$

so the class with highest $P(y|x)$ is predicted. (Note that $P(x)$ is independent of the class and therefore can be omitted.) The naïvety comes from the assumption that in the underlying probabilistic model all the features are independent, and hence $P(x|y) = \prod_{i=1}^{m} P(x_i|y)$. The probabilities $P(x_i|y)$ are derived from

a training set consisting of labelled instances for all possible classes. This is done by a simple counting procedure, often using some smoothing scheme to ensure that all terms appear with non-zero probability. The priors $P(y)$ are often ignored.

In a one-class (OC) setting the training set consists exclusively of normal data. Since a profile of non-self behavior is not required, the detection is performed by simply comparing the probability of a sample being normal (or, equivalently, the anomaly score) to a threshold. Such a threshold can be adjusted to control the false and true positive rates, and the resulting ROC (Receiver Operating Characteristic) curve provides a way of measuring the detection quality.

6.2.2 Instrumentation

The experiments have been conducted in a Google Nexus One smartphone. Power consumption has been measured by applying a battery of tests involving both computation and communication capabilities. Each test is an app containing some of the functionality present in a given anomaly detector, such as for example the training process or the detection stage. The app is loaded into the device and repeatedly executed using some provided configuration. The process is sequential, so only one execution is run at a time.

The device was previously instrumented with AppScope [Yoon et al., 2012] as described in Chapter 3. As mentioned before, AppScope provides the amount of energy consumed by an app in the form of several time series, each one associated with a component of the device (CPU, Wi-Fi, cellular, touchscreen, etc.). We restrict our measures to CPU for computations and Wi-Fi for communications, as our tests

do not have a graphical user interface, do not require user interaction and, therefore, do not use any other component.

6.2.3 Energy Consumption Tests

The energy consumption tests were independently carried out over the four functional tasks described in Section 6.1 in order to obtain a separate consumption model for each anomaly detection component. With this aim in mind, we designed the following four families of tests:

1. *Data preprocessing*. The underlying machine learning algorithm takes as input a dataset of behavioral patterns encoded in some specific format, often in the form of feature vectors. Obtaining such patterns may involve non-negligible computations, such as for example computing histograms, obtaining statistics, applying data transformations, etc. In our case, this stage consisted of processing a trace file where an ordered list of system calls executed by a monitored app was provided. The trace is sequentially read using a sliding window and a feature vector is computed for each window. The vector is then written into an Attribute-Relation File Format (ARFF) file, which will be later used for training or detection purposes. Overall, the preprocessing requires some on-platform computations and also reading and writing files. We used generic I/O Java components for this task, such as FileInputStream and BufferedReader.

2. *Training*. The training process reads an ARFF dataset and builds a model of normal behavior according to some machine learning algorithm. We prepared three different subtests, one for each algorithm discussed above. We used an stripped version of the well known Weka [Hall et al., 2009] library for Android

devices, as this implementation is reasonably optimized. Training involves a number of parameters that may influence the algorithm's running time. In our case, each algorithm was provided with the configuration yielding optimal detection results as discussed in the previous section.

3. *Detection*. This tests measures the amount of energy consumed by a constantly running detector. Again, we prepared one sub-test for each machine learning algorithm and implemented the detector using the stripped version of Weka. Each detector is assumed to have the behavioral model already loaded, so the test only measures energy consumption associated with loading a test vector and deciding its class (normal or anomalous).

4. *Communications*. In this test we measured the amount of energy consumed by sending and receiving data over a Wi-Fi connection. As the amount of data exchanged and the frequency of such exchanges may vary across operational scenarios, we focused on obtaining a model of energy consumed per exchanged byte. We identified three subtests here, depending on whether a secure (encrypted and authenticated) channel is necessary or not. The tests were implemented using standard Java libraries, such as HttpURLConnect and HttpsURLConnect for insecure and secure communications, respectively. Besides, we tested two different networking scenarios. In the first one, the detector communicates with a locally reachable device, which implies low network latency. For these cases we tested both open and WPA-protected Wi-Fi networks. In this case, the time required for a packet to travel from the device to the server and back (Round-Trip Time, RTT) is about 0.6 ms. In the second scenario, we assumed that the detector communicates with a device located reasonably far

Test	Subtest	No. Executions
Data preprocessing	Preprocessing	30
Training	Training.J48	30
	Training.K-means	30
	Training.OCNB	30
Detection	Detection.J48	30
	Detection.K-means	30
	Detection.OCNB	30
Comms	Comms.LoLat.Open.HTTP	30
	Comms.LoLat.Open.HTTPS	30
	Comms.LoLat.WPA.HTTP	30
	Comms.LoLat.WPA.HTTPS	30
	Comms.HiLat.WPA.HTTP	30
	Comms.HiLat.WPA.HTTPS	30

Table 6.1: Energy consumption tests executed.

away in terms of network latency, such as for example in a cloud service accessible via Internet. In our experimental setting, the server is accessed via Internet using a WPA-protected Wi-Fi network with a network latency of around 31 ms.

As indicated above, each test is a separate app that is installed on the device, executed, measured with AppScope, and finally uninstalled. Each test was executed 30 times with different input parameters, such as the length and number of feature vectors in the training dataset and the frequency of sending and receiving data over the network. We elaborate on this later when discussing the experimental results.

The test suite is summarized in Table 6.1.

6.3 Energy Consumption of Anomaly Detection Components

We next present the experimental results obtained after running the tests described in the preceding section. We group the results into two separate categories: computation and communications. The first one includes data preprocessing, training, and detection, while the second focuses on data exchange over the network. We finally obtain and discuss linear regression models for each algorithm and functional task.

6.3.1 Computation

We experimentally found that energy consumption related to preprocessing, training, and detection tasks depends on:

- The length $|v|$ of the feature vectors, measured as the number of attributes that each vector has.
- The size $|\mathcal{D}|$ of the dataset, measured as the number of vectors to be processed, i.e., generated during preprocessing, used for training, or evaluated during detection.

We executed all the preprocessing, training, and detection tests with values of $|v| = 10$, 100, 200, 300, and 400. These lengths are representative of the feature vectors used in most security applications of machine learning. On the other hand, for each vector length we generated datasets of sizes $|\mathcal{D}| = 10$, 50, 100, 200, 500, and 1000, and then computed the average energy consumption per vector. The

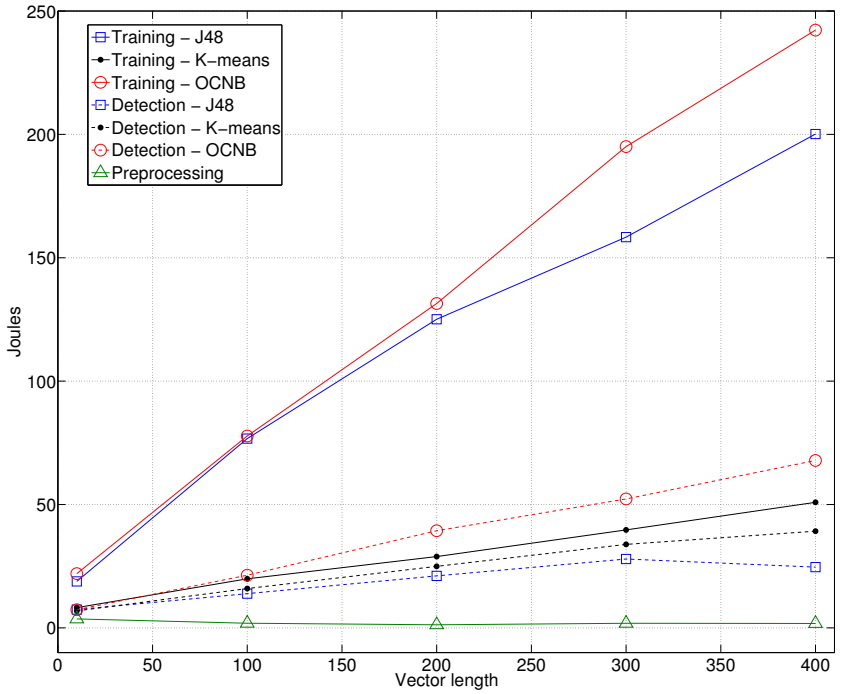


Figure 6.1: Energy consumption results in Joules per vector for different vector lengths for the preprocessing, training, and detection tests.

average energy consumption in Joules (J) per vector for each vector length is shown in Figure 6.1. Several conclusions can be drawn from these results:

1. Data preprocessing consumes very little energy when compared to detection and training. This cannot be easily generalized, as it strongly depends on the sort of preprocessing applied. In our case data preprocessing is quite straightforward (computing histograms) and consumes less than 10 J/vector.

2. For a given algorithm, detection is significantly cheaper than training in terms of energy consumption, but there are exceptions. For example, for both J48 and OCNB, and vectors of length 100 training requires around 50 J/vector more than detection. This difference increases to more than 100 J/vector for

lengths greater than 300. K-means is an exception, with training and detection consuming approximately the same power.

3. The algorithm matters: K-means consumes far less than J48 and OCNB. In turn, OCNB is more expensive power-wise than J48, both in training and detection.

4. For the three tasks, consumption increases approximately linearly in $|v|$.

6.3.2 Communications

Each communication test consists of the app sending and receiving 10 large files to/from a server, using both HTTP and HTTPS. After each test, the total energy consumed is divided by the number of bytes sent or received to obtain a normalized measure in Joules per byte. Each test was repeated 30 times, resulting in the boxplots shown in Figure 6.2.

The results are quite surprising. On the one hand, we found no significant difference between using HTTP or HTTPS. In other words, key establishment plus encryption/decryption for each packet sent/received seems to be extremely efficient in terms of energy consumption. One possible explanation for these figures might be related to the granularity used by AppScope to measure energy and compute the attribution of consumption. AppScope uses application-specific energy consumption data for each hardware component. However, authors argue that the "system" consumes a certain amount of energy when communications are used. It may be the case that AppScope is not attributing the consumption of crypto operations to the app using HTTPS.

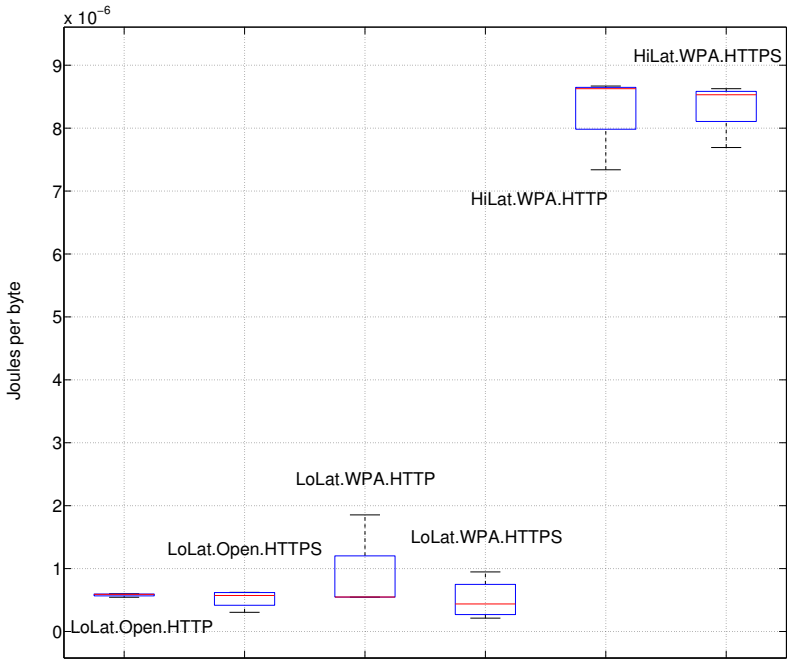


Figure 6.2: Energy consumption results in Joules per byte exchanged (sent or received) for the communication test.

Apart from the observation above, our results suggest that network latency has a clear influence in the consumption. In our experiments, increasing latency from 0.6 ms to 31 ms resulted in 8 times more power. This may just be a consequence of the app execution taking more time to transmit the data.

6.3.3 Linear Models

We used the figures obtained above to derive linear energy consumption models that could be later used to determine the best deployment strategy for each function depending on aspects such as the remaining energy available on the device or the detection architecture. To do this, we applied a simple linear regression analysis using least squares over the energy consumption data.

In the case of the computation functions, each model has the form:

$$E_f(|v|) = \alpha_f \cdot |v| + y_f \tag{6.3}$$

where $f \in \{\texttt{pre}, \texttt{tra}, \texttt{det}\}$, i.e., preprocessing, training, and detection, respectively. Similarly, energy consumption incurred by communications is estimated by a linear model:

$$E_{\texttt{comms}}(nb) = \gamma \cdot nb \tag{6.4}$$

where nm is the number of bytes to be sent or received, and γ is the average energy consumption of the network configuration used by the device.

The coefficients thus estimated are provided in Table 6.2 and confirm the conclusions drawn above. For example the slope α of the three training algorithms reveals the difference between K-means, which introduces a multiplying factor of 0.11 J per additional attribute in the vector, and J48/OCNB, for which such a factor is 0.45 J and 0.57 J, respectively. Similarly, OCNB is clearly much more costly in terms of detection, with a 0.15 J factor per additional vector attribute against 0.05 and 0.08 for J48 and K-means, respectively.

6.4 Deployment Strategies and Trade-offs

Based on the findings presented in the previous section, we next discuss different deployment strategies for the various functions composing an anomaly detection system and analyze the associated energy consumption costs.

Function		Model	
Computation		α_f	y_f
Preprocessing	–	0.00	2.82
Training	Training.J48	0.45	24.45
	Training.K-means	0.11	7.85
	Training.OCNB	0.57	18.78
Detection	Detection.J48	0.05	9.04
	Detection.K-means	0.08	7.16
	Detection.OCNB	0.15	6.34
Communications		γ	
Comms	Comms.LoLat.Open.HTTP	$8.74 \cdot 10^{-7}$	
	Comms.LoLat.Open.HTTPS	$5.09 \cdot 10^{-7}$	
	Comms.LoLat.WPA.HTTP	$5.81 \cdot 10^{-7}$	
	Comms.LoLat.WPA.HTTPS	$5.18 \cdot 10^{-7}$	
	Comms.HiLat.WPA.HTTP	$8.31 \cdot 10^{-6}$	
	Comms.HiLat.WPA.HTTPS	$8.34 \cdot 10^{-6}$	

Table 6.2: Regression coefficients for the linear energy consumption models for computation and communication tasks.

6.4.1 Energy Consumption Strategies

We make two assumptions in our subsequent analysis. Firstly, data acquisition is executed in the device by means of some instrumentation procedure, e.g., through the system API to get access to activity traces. This would not be strictly true for some recently proposed approaches based on keeping a synchronized clone of the device in the cloud [Chun et al., 2011; Portokalidis et al., 2010; Zonouz et al., 2013]. We believe, however, that the overhead incurred by such approaches may be equivalent to that of directly monitoring the device, although this issue needs further investigation. Secondly, our envisioned applications require relatively straightforward data preprocessing (see Table 6.2) that can easily be incorporated into the data acquisition module. As a result, both acquiring the data and preparing the feature

vectors incur a constant overhead for all discussed strategies and will be left out of our analysis.

The two remaining functional blocks are training and detection. Each one, or both, of them can be placed locally in the device (L) or off-loaded to a remote server (R). This gives rise to four possible strategies that will denoted by LL, LR, RL, and RR. In all cases, energy consumption is a linear function:

$$E_{i,j}(t) = \pi_{i,j} \cdot t \tag{6.5}$$

with $i,j \in \{L,R\}$, where $\pi_{i,j}$ is determined by each strategy.

In what follows $|v|$ represents the length in bytes of each feature vector; $|D|$ is the size of the dataset used for training, measured in number of vectors; $|M|$ is the size in bytes of the normality model returned by the training process; and ω_t and ω_d represent the frequencies at which training and detection take place, respectively.

- *Local Training, Local Detection (LL)*. In this case the entire operation of the detector is executed locally in the device. The energy consumption factor π_{LL} is composed of two terms: $E_t(|v|)$ Joules per vector in the dataset during training, plus $E_d(|v|)$ Joules per vector for each detection event. Overall, we have:

$$\pi_{LL} = \omega_t |D| E_t(|v|) + \omega_d E_d(|v|) \tag{6.6}$$

- *Local Training, Remote Detection (LR)*. In this scenario training takes place in the device but detection is off-loaded. During training, energy consumption is equivalent to the corresponding term in (6.6) plus the cost of sending the model M to the cloud ($E_d(|M|)$). In detection mode, every vector must be also

sent out for analysis. We consider here that receiving the result has a negligible cost, as it may just be 1 bit (normal/anomalous). In summary:

$$\pi_{LR} = \omega_t \Big(|D| E_t(|v|) + E_c(|M|) \Big) + \omega_d E_c(|v|) \tag{6.7}$$

- *Remote Training, Local Detection (RL)*. This strategy captures the idea of off-loading the model training stage while performing detection locally. To do this, every time that a (re-)training event is triggered the entire dataset must be sent out for analysis and, subsequently, the model must be received. In detection mode, energy consumption for each analyzed vector is ascribed to the device, resulting in:

$$\pi_{RL} = \omega_t \Big(|D| E_c(|v|) + E_c(|M|) \Big) + \omega_d E_d(|v|) \tag{6.8}$$

- *Remote Training, Remote Detection (RR)*. Finally, this strategy considers the possibility of externalizing all functions to a remote server. Consequently, the only energy consumption attributed to the device is that related to sending and receiving feature vectors both for training and detection. Thus:

$$\pi_{RR} = \omega_t |D| E_c(|v|) + \omega_d E_c(|v|) \tag{6.9}$$

We then discuss the tradeoffs between these four possibilities. In particular, we compare the LL strategy with the other three to understand the potential gains from off-loading training, detection, or both.

6.4.2 LL vs LR

The LL strategy is preferred to LR if:

$$\begin{aligned}
\pi_{LL} &\leq \pi_{LR} \\
\omega_t|D|E_t(|v|)+\omega_d E_d(|v|) &\leq \omega_t\Big(|D|E_t(|v|)+E_c(|M|)\Big)+\omega_d E_c(|v|) \\
\omega_d E_d(|v|) &\leq \omega_t E_c(|M|)+\omega_d E_c(|v|) \\
\omega_d E_d(|v|) &\leq (\omega_t+\omega_d)E_c(|M|+|v|) \\
E_d(|v|) &\leq \frac{\omega_t+\omega_d}{\omega_d}E_c(|M|+|v|) \qquad (6.10)
\end{aligned}$$

Note that, in general, $\omega_d \gg \omega_t$, in which case the term $\frac{\omega_t+\omega_d}{\omega_d} \approx 1$. Alternatively, in the extreme case of training being done for each incoming vector, we have $\omega_d = \omega_t$ and $\frac{\omega_t+\omega_d}{\omega_d} = 2$. Renaming this term as

$$z = \frac{\omega_t+\omega_d}{\omega_d} \in [1,2] \qquad (6.11)$$

and using the linear forms of E_d and E_c we can rewrite the inequality above as:

$$\begin{aligned}
\alpha|v|+y &\leq z\gamma(|v|+|M|) \\
(\alpha-z\gamma)|v| &\leq \gamma|M|-y \\
|v| &\leq z\frac{\gamma|M|-y}{\alpha-z\gamma} \qquad (6.12)
\end{aligned}$$

A simple analysis of the orders of magnitude of the quantities involved in (6.12) provides some insights. Recall that $\alpha \approx 10^{-2}$, $y \approx 10$ and $\gamma \approx 10^{-7}$ (see Table 6.2).

Replacing these values in (6.12), and ignoring the factor z, we get

$$|v| \leq \frac{10^{-7}|M| - 10}{10^{-2} - 10^{-7}} \approx 10^{-5}|M| \tag{6.13}$$

Consequently, the right-hand term in (6.12) will be negative unless $|M|$ is of the order of 10^6 or greater. However, almost all machine learning algorithms produce models that rarely exceed a few hundred kilobytes.

The main conclusion that can be drawn is that the LL strategy is worse energy-wise than the LR unless the model is so large and the vectors tiny enough so that the energy consumed by sending both the model and the vectors to the cloud outweighs the energy of performing detection locally.

6.4.3 LL vs RL

In this case we have:

$$
\begin{aligned}
\pi_{LL} &\leq \pi_{RL} \\
\omega_t|D|E_t(|v|)+\omega_d E_d(|v|) &\leq \omega_t\Big(|D|E_c(|v|)+E_c(|M|)\Big)+\omega_d E_d(|v|) \\
\omega_t|D|E_t(|v|) &\leq \omega_t\Big(|D|E_c(|v|)+E_c(|M|)\Big) \\
|D|E_t(|v|) &\leq |D|E_c(|v|)+E_c(|M|) \\
|D|E_t(|v|) &\leq E_c(|D||v|+|M|) \\
|D|(\alpha|v|+y) &\leq \gamma(|D||v|+|M|) \\
|D|\alpha|v|+|D|y &\leq |D|\gamma|v|+\gamma|M| \\
|D|(\alpha-\gamma)|v| &\leq \gamma|M|-|D|y \\
|v| &\leq \frac{\gamma|M|-|D|y}{|D|(\alpha-\gamma)}
\end{aligned}
\tag{6.14}
$$

Expression (6.14) presents a trade-off somewhat similar to that discussed in the previous section, but more acute. The fact that training takes place remotely factors in the size of the dataset in the inequality, which must be transferred for the remote server to build up the model. The overall consequence is however similar: the RL strategy consumes less than LL unless the model is sufficiently large with respect to the size of the dataset. Since the factor $-|D|y$ appears in the numerator of (6.14), the model size must now be even greater than in the previous case.

In summary, outsourcing the training stage is consistently better than performing it locally unless the datasets to be sent for analysis and the models received are massive.

6.4.4 LL vs RR

Local training and detection consumes less than a fully off-loaded operation if:

$$\begin{aligned} \pi_{LL} &\leq \pi_{RR} \\ \omega_t|D|E_t(|v|)+\omega_d E_d(|v|) &\leq \omega_t|D|E_c(|v|)+\omega_d E_c(|v|) \\ \omega_t|D|\Big(E_t(|v|)-E_c(|v|)\Big) &\leq \omega_d\Big(E_c(|v|)-E_d(|v|)\Big) \end{aligned} \tag{6.15}$$

Note that in (6.15) the various energy consumption functions are applied to inputs of the same length $|v|$. However, communications are several orders of magnitude cheaper than training and detection, so

$$E_t(|v|)-E_c(|v|) \approx E_t(|v|) \tag{6.16}$$

and

$$E_c(|v|)-E_d(|v|) \approx -E_d(|v|) \tag{6.17}$$

Replacing this in (6.15) we get

$$\omega_t|D|E_t(|v|) \leq -\omega_d E_d(|v|) \tag{6.18}$$

which never holds. The conclusion is clear and, in a sense, rather expected from the findings discussed in the two previous sections: off-loading the entire operation of the detector is always better in terms of energy consumption than operating locally in the device.

Taking another look at (6.15), the only scenario where LL may be competitive against RR arises when $E_c(|v|) \geq E_d(|v|)$. This situation may correspond to extremely

lightweight detectors in which computing the anomaly score takes less power than sending the vector over the network. In such a case, (6.15) can be reduced to:

$$|D|\frac{E_t(|v|)}{E_c(|v|)-E_d(|v|)} \leq \frac{\omega_d}{\omega_t} \tag{6.19}$$

which essentially establishes that local operation pays off power-wise if training is very infrequent, does not consume much energy, and the datasets are not very large.

6.4.5 Discussion

The analysis conducted in the previous three sections point out to one definite conclusion: externalizing computation, both training and detection activities, is by far the best option in terms of energy consumption. A deeper look at the trade-offs derived above reveals that the core of this argument is intimately related to the enormous differences in energy consumption existing between computation and networking activities. In platforms such as the current generation of smartphones, communications appear to be extraordinarily optimized in terms of energy requirements, whereas computation is significantly more demanding. In fact, we have seen that communications in current Android devices have similar energy consumption than early wireless sensor devices such as MICAz or TelosB [De Meulenaer et al., 2008]. In the case of applications such as anomaly detection, the best strategy is undoubtedly to externalize all computation functions, including continuous detection, whenever possible.

In terms of performance criteria other than energy consumption, off-loading may or may not have an impact depending on the application domain. Loss of network connectivity—or even sufficient degradation—is a major threat for outsourced detection, as the device may be forced to functioning without the detection service while

the remote server is unreachable. Similarly, network delays may be a critical point in applications where near real-time detection is required. In such cases, these aspects must be weighed against the energy saving benefit.

Offloading resource-intensive tasks to the cloud is a topic that has gained momentum in recent years. Several works (e.g., [Kumar and Lu, 2010; Namboodiri and Ghose, 2012; Tandel and Venkitachalam, 2013]) have addressed the issue of deciding whether *to cloud* is a better option than *not to cloud* for mobile systems. For example, in [Namboodiri and Ghose, 2012] it is shown that determining an energy efficient strategy is a complex task and require a fine characterization of the impact of several parameters, including the type of device and the application domain. Their approach focuses on three rather generic applications: word processing, multimedia and gaming for both laptops and mobile devices. Authors conclude that "cloud-based applications consume more energy than non-cloud ones" when using platforms such as mobile devices. In contrast, other works such as [Tandel and Venkitachalam, 2013] and [Kumar and Lu, 2010] show that offloading is generally profitable energy-wise, particularly for intensive computation tasks that require relatively small amount of communications.

Finally, the security and privacy aspects of offloading computation to the cloud is a major concern that may prevent many users from relying on external services, particularly when confidential data is involved in the training and detection datasets. In this context, many works have dealt with the problem of securely outsourcing computation (see, e.g. [Wang et al., 2011]). One common assumption is to consider the external server as untrusted and to encrypt all data sent out for processing. In order to assess the extra energy consumption incurred by encrypting data prior to sending it, we evaluated three of the most common ciphers found in cryptographic

Cipher	Mode	γ
AES-128	CTR	$7.62 \cdot 10^{-9}$
3DES	CTR	$9.52 \cdot 10^{-9}$
RC4	–	$7.62 \cdot 10^{-9}$

Table 6.3: Average energy consumption per encrypted byte.

libraries and used nowadays: AES, 3DES, and RC4. The experimental setting and energy consumption tests are identical to those described in Section **??**. We carried out 30 independent tests and divided, in each case, the total energy consumed by the number of encrypted bytes to obtain a normalized measure in Joules per byte. The γ factor obtained is shown in Table 6.3. As it can be observed, the cost of encryption is negligible when compared to that of training and detection tasks and does not affect the general conclusions discussed above. These results indicates that current encryption algorithms are extremely efficient in terms of energy consumption. This is further supported by the results shown in the literature during the last years aiming at providing low budget cryptography to enable wireless security [De Meulenaer et al., 2008; Fan et al., 2013; Karaklajiić et al., 2010; Kerckhof et al., 2012; Verbauwhede, 2011].

6.5 Case Study: A Detector of Repackaged Malware

We next illustrate some of the conclusions drawn in the preceding sections with real-world application: an anomaly-based detector for repackaged malware in Android apps. The use of anomaly detectors for this purpose has been proposed in a number of recent works (see, e.g., [Burguera et al., 2011; Shabtai et al., 2012]). Although in all cases the performance of such approaches is reasonably good in terms of detection

quality, to the best of our knowledge none has explored the energy consumption savings gained by outsourcing it.

6.5.1 The Detector

Sequences of system calls have been recurrently used by anomaly detection systems for security applications in smartphones [Blasing et al., 2010; Burguera et al., 2011; Lin et al., 2013; Shabtai et al., 2012]. All apps interact with the platform where they are executed by requesting services through a number of available system calls. These calls define an interface that allow apps to read/write files, send/receive data through the network, read data from a sensor, make a phone call, etc. Legitimate apps can be characterized by the way they use such an interface [Lin et al., 2013], which facilitates the identification of malicious components inserted into an seemingly harmless app and, more generally, other forms of malware [Suarez-Tangil et al., 2014b].

Based on this idea, we have built an anomaly detector that combines some of the ideas already proposed in previous works[1]. Feature vectors consist of histograms computed from a trace of system calls using a sliding window of length W. We determined experimentally that windows of length 400 result in very good detection performance. The number of systems calls varies across architectures and it is often between 200 and 400. Thus, during the training period all processes of normal apps are monitored and the corresponding feature vectors are generated. Such vectors are then used to train a normality model.

In detection mode, the algorithm takes as input a sequence sq of N system calls

[1]We deliberately omit a number of details about our detector, particularly those related to the detection quality for different parametrizations, as this is not the main focus of this work and has been reported elsewhere.

and extracts the $N - W + 1$ feature vectors using a sliding window. Each one of these feature vectors is then classified as normal or anomalous. Let A be the number of vectors identified as anomalous. Then, the sequence—and, therefore, the app—is classified according to the following rule:

$$\mathsf{det}(sq) = \begin{cases} \mathsf{legitimate} & \text{if } \frac{A}{N-W+1} < \rho \\ \mathsf{repackaged} & \text{otherwise} \end{cases} \tag{6.20}$$

where ρ is an adjustable detection threshold.

The detection procedure described above is intimately related to the nature of repackaged malware. In general, not all the system call windows issued by a repackaged app will be anomalous, as they may be generated by non-malicious code. Thus, detection must be based on analyzing sets of windows and seeking if a fraction of them are anomalous. In our experiments, we obtained good results with sequences of at least 10 windows and thresholds ρ around 0.1. For example, one of the apps we used for testing detection performance is a popular game named *Mx Moto* by Camel Games. The app can be purchased from Google Play for 1.49 €and so far has been downloaded 100K times. The same app can also be found in alternative markets for free [Zhou and Jiang, 2012], in most cases repackaged with a malware known as *Anserverbot*. We tested the original app together with various repackaged variants, obtaining in all cases a detection rate of 100% with no false positives with the OCNB detector. These results are congruent with those reported in similar works based on anomaly detection [Burguera et al., 2011; Shabtai et al., 2012].

Type	No. Apps	No. Events	Throttle (ms)	Syscalls/s
Goodware	10	5000	1000	180.43
	35	1000	5000	453.91
	50	5000	1000	307.62
Malware	10	5000	1000	112.39
	35	1000	5000	128.66
	50	5000	1000	161.90
All	**190**	—	**Average**	**224.16**

Table 6.4: Average number of system calls per second in different executions of both goodware and malware.

6.5.2 Testing Framework

We tested the energy consumption of three detectors built as described above, one for each machine learning algorithm evaluated. Only the LL and RR strategies were studied, as they represent opposite cases for placement decisions. For the latter, the high latency configuration with WPA and HTTPS was used. In order to study energy consumption for different apps and/or detector configurations, we gathered a dataset of 190 apps containing both goodware and malware. For each one of them, we derived the average number of system calls per second issued depending on different usage intensity rates (throttle). These figures are obtained by running each app in a controlled environment and automatically injecting user events at a throttle pace. The results are shown in Table 6.4 and reveal that apps can generate up to a few hundred system calls per second. Even though user-driven apps may well function at lower paces, these rates are useful for apps where high frequency testing is required.

Each detector is evaluated for different vector lengths. Again, our goal is measuring how the amount of energy varies in a real setting depending on the choice of this parameter. (Recall that in terms of detection quality, best results are obtained

for $|v| = 100$.) Finally, each detector was continuously executed during 1 week, and the amount of energy consumed so far was measured at 4 control points: after 10 minutes, 1 hour, 1 day, and 1 week. During this period, detection is triggered as often as a sufficiently large sequence of system calls is available, and re-training occurs every 10 minutes.

6.5.3 Results and Discussion

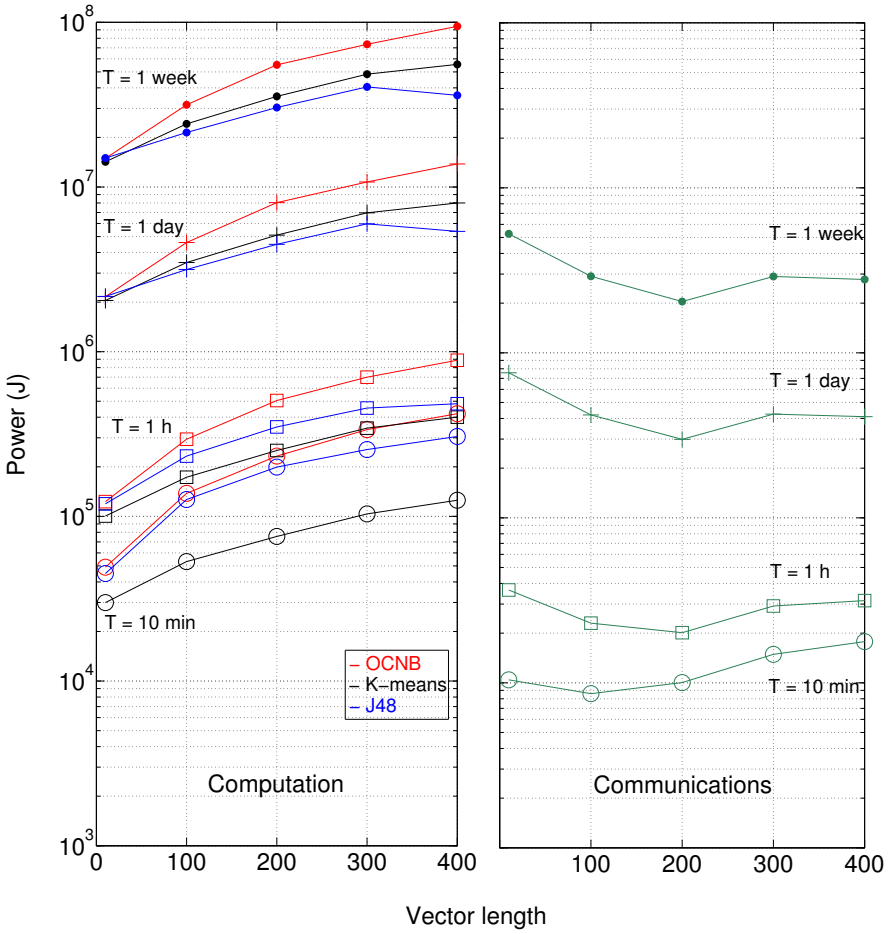


Figure 6.3: Average energy consumption for different detectors using the LL (Computation) and RR (Communications) strategies.

Figure 6.3 shows the average energy consumed by the three detectors for the LL and RR strategies. (Note that the latter is independent of the algorithm as only communications are involved.) The plots are consistent with the results discussed in the previous section and confirm that outsourced detection is much more efficient energy-wise than on-platform operation. Consider, for example, the case of vectors of 100 attributes. During the first 10 minutes, both the OCNB and the J48 detectors have consumed more than 10^5 J. During the same period, the detector located in the cloud has required less than 10^4 J. After 1 day, cloud-based detectors consume roughly the same amount of energy than on-platform detectors over 1 hour. Note, too, that the frequency of re-training is extremely high in this setting, and that the difference would be substantially greater if training occur more sporadically.

Another interesting finding is that differences among algorithms are noticeable after some time, especially for large vectors. In general, OCNB is much more demanding than K-means and J48 when vectors with a hundred attributes are involved.

Finally, in order to contextualize the energy implications of constantly running a detector, we have measured the energy consumed by some popular apps during 10 minutes (see Table 6.5). These apps are representative of three broad classes of popular activities: games, online social networking, and multimedia content. The amount of energy consumed by the three ranges between approximately 550 J and 645 J, most of it being related to the graphical user interface. For comparison purposes, running our detector in the device with the less demanding algorithm (J48) takes around 15 J per detection. At full throttle (i.e., around 224 detections per second) this implies a consumption of around 2 MJ in 10 minutes. Even if detection only takes place at a rate of 1 per second, the overall consumption in 10 minutes is still around 9 KJ. In contrast, outsourced detection using WPA, HTTPS, and high

App	CPU	Comms	Display	Total
YouTube	30.11	12.59	508.90	551.59
MX Moto	129.24	5.75	509.54	644.52
Facebook	137.76	27.42	471.42	637.27

Table 6.5: Consumption (in Joules) of three popular apps during a time span of 10 minutes.

latency consumes around 112 J and 0.5 J in the same conditions, respectively.

The figures discussed above reinforce the conclusion that externalized operation of anomaly detection seems to be the only reasonable choice in terms of energy consumption. However, given that cloud-based processing may raise some privacy concerns in certain applications, this also motivates the need for more lightweight anomaly detection techniques that may be suitable for on-platform operation.

6.6 Conclusions

In this chapter, we have discussed the power consumption trade-offs among various strategies for executing anomaly detection components directly on mobile platforms or remotely in the cloud. Both our theoretical analysis and experimental results confirm that there is actually little choice but to offload everything to the cloud. Reasons for this include the differences between the energy efficiency of computation and communications in current platforms, and also various parameters related to the anomaly detection setting, such as the dataset sizes and the operation frequency. We believe that the linear models provided in this work may be useful in other contexts to obtain estimates about the energy consumption of different alternatives. Furthermore, such models can be easily extended to other machine learning algorithms by simply deriving the appropriate coefficients α and y.

7

Detecting Targeted Smartphone Malware with Behavior-Triggering Stochastic Models

7.1 Introduction

Malware for smartphones is a problem that has rocketed in the last few years [Juniper, 2013]. The presence of increasingly powerful computing, networking and sensing functions in smartphones has empowered malicious apps with a variety of advanced capabilities [Suarez-Tangil et al., 2014b], including the possibility to determine the physical location of the smartphone, spy on the user's behavioral patterns, or compromise the data and services accessed through the device. These capabilities are rapidly giving rise to a new generation of *targeted* malware that makes decisions on the basis of factors such as the device location, the user's profile, or the presence of other apps (e.g., see [Felt et al., 2011c; Hasan et al., 2013; Raiu and Emm, 2013; Zawoad et al., 2013]). The idea of behaving differently under certain circumstances

was also successfully applied in the past. For instance, Stuxnet [Langner, 2011] remained dormant until a particular app was installed and used at certain location, having as a target Iranian Nuclear Plants. Other malware targeted governments and private corporations—mostly in the financial and pharmaceutical sectors [Corporation, 2013]. Another representative example of targeted malware is Eurograbber [Kalige and Burkey, 2012], a "smart" Trojan targeting online banking users. The situational awareness provided by smartphone platforms makes this type of attacks substantially easier and potentially more dangerous. More recently, other examples of targeted malware include FinSpy Mobile [Marquis-Boire et al., 2013], a general surveillance software for mobile devices, and Dendroid Remote Access Toolkit (RAT) [Rogers, 2014], which offers capabilities to target specific users.

A similar problem is the emergence of the so-called *grayware* [Felt et al., 2011c], i.e., apps that cannot be completely considered malicious but whose behavior may entail security and/or privacy risks of which the user is not fully aware. For example, many apps using targeted advertisements are particularly aggressive in the amount of personal data they gather, including sensitive contextual information acquired through the device sensors. The purpose of such data gathering activities is in many cases questionable, and many users might well disapprove it, either entirely or in certain contexts.[1]

Both targeted malware and grayware share a common feature that complicates their identification: the behavior and the potential repercussions of executing an app might depend quite strongly on the context where it takes place [Capilla et al., 2014] and the way the user interacts with the app and the device [Gianazza et al.,

[1] Classical examples include two popular games, *Aurora Feint* and *Storm8*, which were removed from the Apple Store for harvesting data and phone numbers from the user's contact list and sending them to unknown destinations as introduced in Chapter 2.

2014]. We stress that this problem is not addressed by current detection mechanisms implemented in app markets, as operators are overwhelmed by the number of apps submitted for revision every day and cannot afford an exhaustive analysis over each one of them [Chakradeo et al., 2013]. A possible solution to tackle this problem could be to implement detection techniques based on dynamic analysis (e.g., Taintdroid [Enck et al., 2010]) directly in the device. However, this is simply too demanding for battery-powered platforms. Several recent works [Chun et al., 2011; Kosta et al., 2012; Portokalidis et al., 2010; Zonouz et al., 2013] have proposed to keep a synchronized replica (clone) of the device virtualized in the cloud. This would facilitate offloading resource-intensive security analysis to the cloud, but still does not solve one fundamental problem: grayware and targeted malware instances must be provided with the user's particular context and behavior, so the only option left would be to install the app, use it, and expect that the analysis conducted over the clone—hopefully in real time—detects undesirable behaviors. This is a serious limitation that prevents users from learning in advance what an app would do in certain situations, without the need of actually reproducing such a situation.

Recent works such as PyTrigger [Fleck et al., 2013] have approached the problem of detecting targeted malware in Personal Computers (PC). To do so, it is sought to trigger specific malware behaviors by injecting activities collected from users (e.g., mouse clicks and keyword inputs) and their context. This approach cannot be adopted to platforms such as smartphones because the notion of *sensed* context is radically different here. Other schemes, including the work presented in [Gianazza et al., 2014; Jensen et al., 2013; Rastogi et al., 2013a; Zheng et al., 2012], do focus on smartphones but concentrate exclusively on interactions with the Graphical User Interface (GUI) and are vulnerable to context-based targeted attacks.

Two works closer to our proposal are Context Virtualizer [Liang et al., 2013] and Dynodroid [Machiry et al., 2013], where a technique called context fuzzing is introduced in the former and used in the latter. The main aim in [Liang et al., 2013; Machiry et al., 2013] is to automatically test apps with real-world conditions, including user-based contexts. These tools, however, are intended for developers who want to learn how their apps will behave when used in a real setting. Contrarily, our focus is on final users who want to find out if they will be targeted by malicious or privacy-compromising behaviors. Finally, other works such as CopperDroid [Reina et al., 2013] focus on malware detection as we do, but with a static approach (based on information extracted from the manifest) that, besides, does not consider the user context.

In this chapter, we address the problem of identifying targeted grayware and malware and propose a more flexible approach compared to other proposals to determining whether the behavior of an app is compliant with a particular set of security and privacy preferences associated with an user. Our solution is based on the idea of obtaining an *actionable* model of user behavior that can be leveraged to test how an app would behave should the user executes it in some context. Such a testing takes place over a clone of the device kept in the cloud. This approach removes the need of actually exposing the device (e.g., we let the device access only fake data and not real). More importantly, the analysis is tailored to a given user, either generally or for a particular situation. For example, a user might want to explore the consequences of using an app in the locations visited during working days from 9 to 5 or during a planned trip.

Section 7.2 introduces the theoretical framework used to model triggering patterns and app behavior. In Section 7.3, we describe the architecture of our proposal

and a proof-of-concept prototype, and discuss the experimental results obtained in terms of testing coverage and efficiency. In Section 7.4, we discuss the detection performance with two representative case studies of grayware and targeted malware instances. Finally, Section 7.5 extracts some conclusions.

7.2 Behavioral Models

This section introduces the theoretical framework used in our proposal (presented in Section 7.4) to trigger particular app behaviors and determining whether they entail security risks to the user (as shown in Section 7.4). More precisely, we present models for the user-provided inputs, the resulting app behavior, and the mechanism used to assess potential risks.

7.2.1 Triggering Patterns

Inputs provided by the user to his device constitute a major source of stimuli for triggering certain app behaviors. We group such inputs into two broad classes of patterns, depending on whether they refer to inputs resulting from the user directly interacting with the app and/or the device (e.g., through the touchscreen), or else indirectly by the context (e.g., location, time, presence of other devices in the surroundings).

7.2.1.1 Usage Patterns

Usage patterns model sequences of events resulting from the actions of the user during his interaction with an app. Such events are internal messages passed on to

the app by the device, such as starting an activity, or clicking a button. We underline that our focus is on the events and not on the actions that generate them, as the same event can be triggered through different input interfaces (e.g., touchscreen, voice). Let the following be a set of all possible events for all apps:

$$\mathcal{B} = \{e_1, e_2, \ldots, e_n\}. \tag{7.1}$$

Thus, the interaction of a user with an app can be represented as an ordered sequence:

$$\mathbf{u} = \langle \varepsilon_1, \varepsilon_2, \ldots, \varepsilon_k \rangle, \quad \varepsilon_i \in \mathcal{B}. \tag{7.2}$$

We will refer to such sequences as *usage traces*. Interactions with an app at different times and/or with different apps will result in different usage traces.

7.2.1.2 Context Patterns

Apps may behave differently depending on conditions not directly provided by the user, such as the device location, the time and date, the presence of other apps or devices, etc. We model this using the widely accepted notion of *context* [Conti et al., 2012]. Assume that $v_1, \ldots, v_m$ are variables representing contextual elements of interest, with $v_i \in \mathcal{V}_i$. Let the following be the set of all possible contexts:

$$\mathcal{X} = \mathcal{V}_1 \times \cdots \times \mathcal{V}_m. \tag{7.3}$$

As above, monitoring a user during some time interval will result in a sequence refer to as *context traces*:

$$\mathbf{g} = \langle x_1, x_2, \ldots, x_l \rangle, \quad x_i \in \mathcal{X}. \tag{7.4}$$

7.2.2 Stochastic Triggering Model

Usage and context traces are used to derive a model that captures how the user interacts with an app or a set of apps. For this purpose, we rely on a discrete-time first-order Markov process (i.e., a Markov chain [Norris, 1998]) $\mathbf{M} = (S, A, \Pi)$ where:

- The set of states S is given by:

$$S = \mathcal{B} \times \mathcal{X} = \{s_1, \ldots, s_N\}. \tag{7.5}$$

  We will denote by $q(t) \in S$ the state of the model at time $t = 1, 2, \ldots$, representing one particular input event executed in a given context.

- The transition matrix is given by:

$$A = [a_{ij}] = P[q(t+1) = s_j | q(t) = s_i], \tag{7.6}$$

  where $a_{ij} \in [0, 1]$ and $\sum_{j=1}^{N} a_{ij} = 1$.

- The vector of initial probabilities is given by:

$$\Pi = (\pi_i) = P[q(1) = s_i], \tag{7.7}$$

with $\pi_i \in [0,1]$ and $\sum_{i=1}^{N} \pi_i = 1$.

The model above is simple yet powerful to model user-dependant behavioral patterns when interacting with an app. The model parameters can be easily estimated from a number of usage and context traces. Assume that $\mathbf{O} = \{o_1, o_2, \ldots, o_T\}$ is a sequence of observed states (i.e., event-context pairs) obtained by monitoring the user during a representative amount of time. The transition matrix can be estimated as:

$$a_{ij} = \frac{\sum_{t=2}^{T} P[q(t) = s_j | q(t-1) = s_i]}{\sum_{t=2}^{T} P[q(t) = s_j]} = \frac{\sum_{t=2}^{T} P[o_t = s_j | o_{t-1} = s_i]}{\sum_{t=2}^{T} P[o_t = s_j]}, \tag{7.8}$$

where both probability terms are obtained by simply counting occurrences from $\mathbf{O}$. The process can be trivially extended when several traces are available.

The model above should be viewed as a general modeling technique that can be applied at different levels. Therefore, if one is interested in modeling input events irrespective of context, the set of states—and, therefore, the chain—can be reduced to $\mathcal{B}$. The same applies to context; e.g., states could be composed exclusively of time-location pairs.

Markov chains are often represented as a directed graph where vertices represent states and edges between them are labelled with the associated transition probability. We will call the *degree* of a state, denoted by $\mathbf{deg}(s_i)$, to the number of states reachable from s in just one transition with non-null probability

$$\mathbf{deg}(s_i) = \#\{p_{ij} | p_{ij} > 0\}. \tag{7.9}$$

The degree distribution of a chain is given by:

$$\mathbf{P}(k) = P[\mathbf{deg}(s) = k]. \tag{7.10}$$

7.2.3 App Behavior and Risk Assessment

An app interacts with the device by requesting services through a number of available system calls. These define an interface for apps that need to read/write files, send/receive data through the network, make a phone call, etc. Rather than focusing on low-level system calls, in this chapter we will describe an app behavior through the sequence of *activities* it executes (see Chapter 5-Section 5.2.2). Activities represent high-level behaviors, such as for example reading from or writing into a file, opening a network connection, sending/receiving data, etc. In some cases, there will be a one-to-one correspondence between an activity and a system call, while in others an activity may encompass a sequence of system calls executed in a given order. In what follows, we assume that

$$\mathbb{A} = \{a_1, a_2, \ldots, a_r\} \tag{7.11}$$

is the set of all relevant activities observable from an app execution.

The execution flow of an app may follow different paths depending on the input events provided by the user and the context. Let $\boldsymbol{\sigma} = \langle \boldsymbol{\sigma}_1, \ldots, \boldsymbol{\sigma}_k \rangle$ be a sequence of states as defined above. We model the behavior of an app when executed with $\boldsymbol{\sigma}$ as

input as the sequence

$$\beta(\sigma) = \langle \alpha_i, \ldots, \alpha \rangle, \quad \alpha_i \in \mathbb{A}, \tag{7.12}$$

which we will refer to as the *behavioral signature* induced by σ.

Behavioral signatures constitute dynamic execution traces generated with usage and context patterns specific to one particular user. Analysis of such traces will be instrumental in determining whether there is evidence of security and/or privacy risks for that particular user. The specific mechanism used for that analysis is beyond the scope of our current work. In general, we assume the existence of a *Risk Assessment Function* (RAF) implementing such a analysis. For example, general malware detection tools based on dynamic analysis could be a natural option here. The case of grayware is considerably more challenging, as the user's privacy preferences must be factored in to resolve whether a behavior is safe or not.

7.3 Targeted Testing in the Cloud

In this section, we first describe the architecture and the prototype implementation of a cloud-based testing system for targeted malware and grayware based on the models discussed in the previous section. We then provide a detailed description of various experimental results obtained in two key tasks in our system: obtaining triggering models and using them to test a cloned device.

7.3.1 Architecture and Prototype Implementation

A high level architectural view of our system is shown in Figure 7.1. There are two differentiated major blocks: (i) the *evidence generation* subsystem, and (ii) the

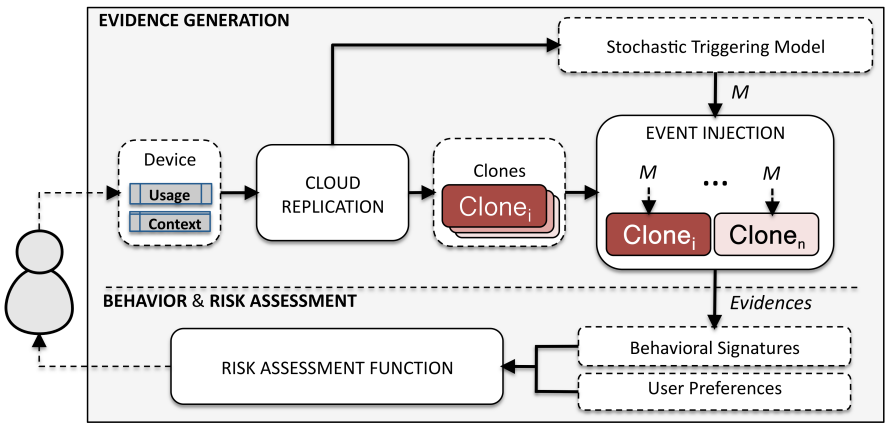


Figure 7.1: System architecture and main building blocks.

behavioral modeling and risk assessment subsystem. The first one extracts usage and context traces from the device and generates the stochastic triggering model. This process is carried out by first cloning the user device into the cloud and then injecting the triggering patterns over the clone. The second block extracts the behavioral signatures from the clone(s) and applies the RAF over the evidences collected. We next provide a detailed description of our current prototype implementation.

The experiments have been conducted over the laboratory described in Chapter 3. Specifically, we instrumented both a physical device and a cloud-virtual device. We inject events and contexts into apps and monitor the resulting behavior. We chose 20 relevant activities to characterize app behavior (see Table 7.1), which include information about calls to the crypto API (`cryptousage`), I/O network and file activity (`opennet`, `sendnet`, `accessedfiles`, etc.), phone and SMS activity (`phonecalls`, `sendsms`), data exfiltration through the network (`dataleak`), and dynamic code injection (`dexclass`), among others.

Finally, we implemented a simple yet powerful RAF (*Risk Assessment Function*) for analyzing behavioral signatures. In essence, the scheme is based on a pattern-

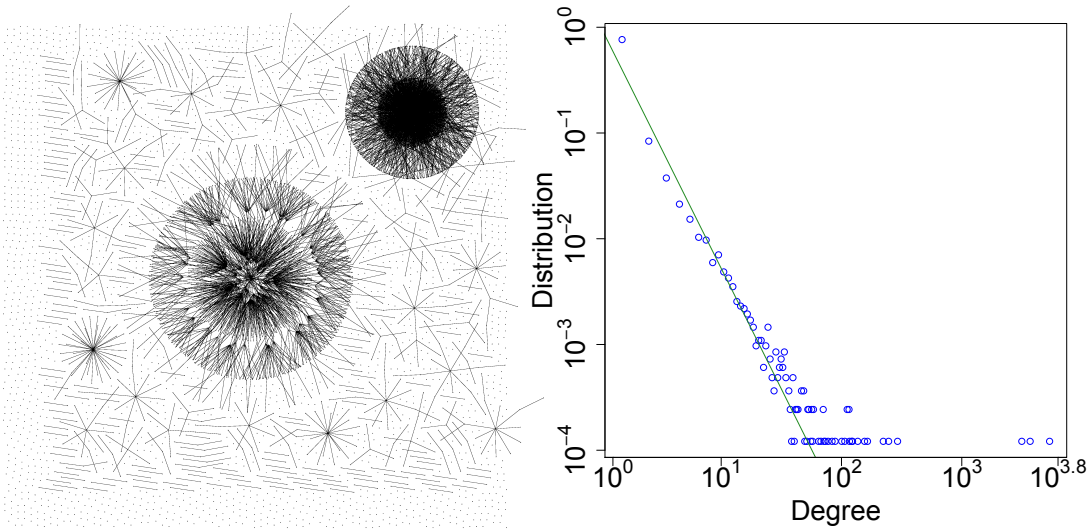
Activities				
• sendsms	• servicestart	• phonecalls	• udpConn	• cryptousage
• sendnet	• netbuffer	• activities	• dexclass	• activityaction
• dataleak	• enfperm	• opennet	• packages	• permissions
• recvs	• recvnet	• recvsaction	• fdaccess	• accessedfiles

Table 7.1: Set of activities ($\mathbb{A}$) monitored from an app execution and used to characterize its behavior.

matching process driven by a user-specified set of rules that identify behaviors of interest according to his security and privacy preferences. Such rules are first-order predicates over the set of activities $\mathbb{A}$, allowing the user to specify relatively complex patterns relating possible activities in a signature through logical connectives. Regardless of this particular RAF, our prototype supports the inclusion of standard security tools such as, e.g., antivirus packages or other security monitoring components. These can be easily uploaded to the clone and run while the testing carries on.

7.3.2 Experiment I: The Structure of a Triggering Model

In this first experiment, we monitored all events triggered by a user executing several apps on his device during a representative amount of time. The resulting event set contained about $|S|$ =8K states, distributed over various observations traces of around $|\mathbf{O}| = 37$K states. We then used such traces to estimate the transition matrix using Eq. (7.8). The resulting Markov chain turned out to have various interesting features. For example, its degree distribution follows a power-law of the form $\mathbf{P}(k) = k^{-\alpha}$ (see Fig. 7.2) with $\alpha = 2.28$ for $k \geq 2$. This suggests that events and contexts follow a scale-free network [Clauset et al., 2009], which is not surprising. Recall that an edge between two nodes (events) indicates that the destination event



(a) (b)

Figure 7.2: (a) Markov model representing contextual and kernel input events for a user interacting with an Android platform; (b) Degree distribution, in log-log scale, of the model in (a) as defined in Section 7.2.2.

occurs after the source event.

A power-law distribution such as the one shown in Figure 7.2 reveals that most events have an extremely low number of "neighbors"; i.e., once an event has happened, the most likely ones coming next reduce to about 100 out of the 8K possible. Only a small fraction of all events are highly connected, meaning that almost any other event is possible to occur after them. For instance, in our traces we found that over half of the states were only connected to just one state. In contrast, one state was found to be connected to more than 4000 other states.

These results make sense due to the following reason: input and context events do depend quite strongly on those issued immediately before. For example, the probability of moving from one place to another nearby is much higher than to a remote place. The same applies to sequences of events, where the probability distribution of the next likely event reflects the way we interact with the app. As we

will next see, this structure makes testing extremely efficient.

7.3.3 Experiment II: Speed of Testing

We performed a number of experiments to measure how fast input events can be injected into an Android application sandbox. Such events include not only input events, but also a variety of contexts' traces comprising phone calls, SMS messages and GPS locations. We studied the time taken by both the sandbox and the operating system to process each injected event. Our results suggest that the time required to process injected states (input or context events) varies depending on the type of state (see Table 7.2). For instance, it takes around 0.35 seconds, on average, to inject an SMS and process it trough the operating system. In contrast, geolocation events can be injected almost 100 times faster. We also observed a significant difference between the capabilities of the sandbox and the OS running on top of it. For instance, while the sandbox is able to process about 2800 geolocation states per second, the OS can only absorb around 100 each second. We suspect that this throughput might be improved by using more efficient virtual frameworks, such as Qemu for Android x86[2] or ARM-based hardware for the cloud.[3]

For comparison purposes, the lower rows in Table 7.2 show the average and peak number of events generated by human users, both for usage (e.g., touch events) and context events, as reported in previous works [Wei et al., 2012].

[2] http://www.android-x86.org/
[3] http://armservers.com/

Automatic Injection		
Injected Event	**Emulator Layer**	**App Layer**
Sensor event	7407.66 events/s	1.26 events/s
Power event	361.77 events/s	19.16 events/s
Geolocation event	2810.15 events/s	111.87 events/s
SMS event	451.27 events/s	0.35 events/s
GSM call/cancel event	1726.91 events/s	0.71 events/s

Human Generated		
Event Type	**Average**	**Peak**
Usage patterns	5 events/s	10 events/s
Context patterns	10 events/s	25 events/s

Table 7.2: Event injection rates for different types of events over a virtualized Android device (top), and rates generated by real users based on profiling 67 apps [Wei et al., 2012] (bottom).

7.3.4 Experiment III: Coverage and Efficiency

We perform a number of experiments to evaluate the performance of our proposal. We aim at measuring the time required to reach an accurate decision by means of simulation. More precisely, we simulate an injection system configured with an specific $\mathbf{u}$ and $\mathbf{g}$ randomly generated and with different number of states $|S| = 100, 1000, 10000$.

The configuration of each experiment is based on the findings shown in previous section as detailed bellow. First, we generated two types of Markov model chains: (i) one random scale-free network of events using a preferential attachment mechanism as defined by *Barabási–Albert* (BA) [Albert and Barabási, 2002], and (ii) another random network with attachment mechanism as defined by *Erdős-Rényi* (ER) model [Erdős and Rényi, 1960]. Then, we simulated a user providing inputs to a device together with its context at a rate of 10 events per second. We chose this throughput as it is a realistic injection rate (see Table 7.2).

In each experiment, we generate a number of random Markov chains and calcu-

late the cumulative transition probability covered when traversing from one state to another of the chain for the first time. Formally, let

$$rw = \langle s_{i_1}, s_{i_2}, \ldots, s_{i_n} \rangle, \qquad s_{i_j} \in S \tag{7.13}$$

be a random walk over the chain, with $a_{i_j i_{j+1}} > 0 \ \forall i_j$, and let

$$T(rw) = \{(s_{i_j}, s_{i_{j+1}}) \mid s_{i_j} \in S \setminus \{s_{i_n}\}\} \tag{7.14}$$

be the set of all transitions made during the random walk. We define the coverage of *rw* as the amount of transitions seen by *rw*, weighted by their respective probabilities and normalized to add up to one, i.e.:

$$\texttt{Coverage}(rw) = \frac{1}{N} \sum_{(p,q) \in T(rw)} a_{pq}. \tag{7.15}$$

The coverage is used to evaluate both the efficiency and the accuracy of our system. On the one hand, it can be used to measure the amount of a user's common actions triggered given a limited period of testing time. Additionally, it also shows how fast the system tests the most common actions. Results for sets of events of various sizes are shown in Figure 7.3, where the curves have been averaged over 10 simulations. The results show that the coverage reached when testing networks of sizes $|S| = 100$, 1000, and 4000 states is very satisfactory. Such a good performance is related to the scale-free distribution of states through time. Thus, a coverage above 80% is reached in less than two minutes for 100 states, and in approximately 1 hour for 4000 states.

It is important to emphasize that the coverage reported in Figure 7.3 corresponds

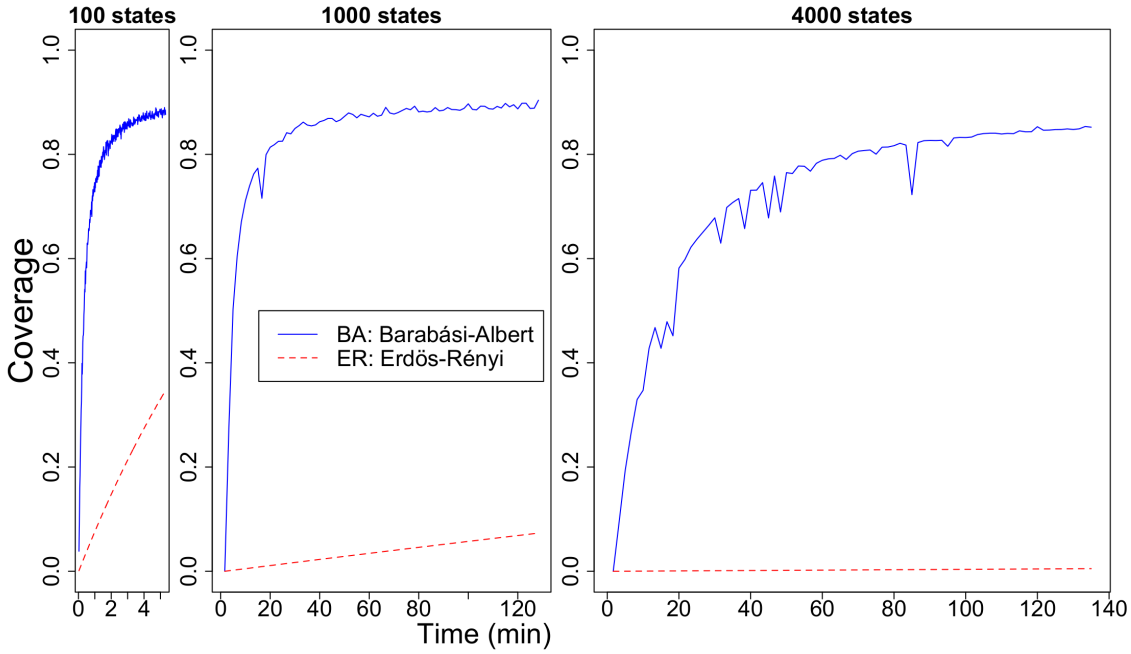


Figure 7.3: Efficiency and accuracy of the decision for a Barabási-Albert and Erdős-Rényi network model.

to one test sequence randomly drawn according to the user's behavioral model. If the process is repeated or carried out in parallel (e.g., over a clone), other test sequences may well explore behaviors not covered by the first one. This is illustrated in Table 7.3, where we show the total testing coverage as a function of the number of clones tested in parallel, each on with a different input sequence. Thus, after two hours testing just one clone results in a coverage slightly above 84%. However, if five clones are independently tested in parallel, the overall results is a coverage of around 93% of the total user behavior. This time-memory trade-off is a nice property, allowing to increase the coverage by just testing multiple clones simultaneously rather than by performing multiple test over the same clone.

Reaching a 100% coverage is, in general, difficult due to the stochastic nature of the models. This is not critical, as those behavioral patterns that are left unexplored correspond to actions extremely unlikely to be executed by the user. In practical terms

	Number of parallel clones									
	1	2	3	4	5	6	7	8	9	10
10 min.	42%	60%	68%	73%	76%	79%	81%	81%	82.5%	83.4%
60 min.	79%	86%	89%	90%	90%	91%	91%	91%	91%	95%
120 min.	84%	87%	88%	88%	93%	93%	93%	93%	93%	93%

Table 7.3: Coverage given by our model when running multiple parallel clone given a limited testing time for a network of $|S| = 4000$ states.

this is certainly a risk, but one relatively unimportant as the presumably uncovered malware instance would not activate for this user except with very low probability.

7.4 Case Studies

In this section we present two case studies illustrating how the injection of user-specific behavioral patterns can contribute to reveal malware with targeted activation mechanisms. We cover *dormant* and *anti-analysis* malware, as these scenarios constitute representative cases of targeted behaviors in current smart devices [Suarez-Tangil et al., 2014b]. For each case, we first provide a brief description of the rationale behind the malware activation condition and then discuss the results obtained after applying the injection strategy presented in this work. In all cases, the evaluation has been conducted using the android remote access tool (RAT) described in Chapter 3. More precisely, we have adapted *Androrat* [Bertrand et al., 2014] to incorporate the specific triggering conditions.

7.4.1 Case 1: Dormant Malware/Grayware

Piggybacked malware [Zhou et al., 2013] is sometimes programmed to remain dormant until an specific situation of interest presents itself [Zhou and Jiang, 2012].

Wake-up conditions	
User presence	USB connected, screen-on action, accelerator changed, etc.
Location	Location change event, near an address, leaving an area, etc.
Time	A given day and time, after a certain period of time, etc.
Hardware	Power and LED status, KEY action, LOCK event, etc.
Configuration	Apps installed, a given contact/phone number in the agenda, etc.

Table 7.4: Typical wake-up conditions for malware activation.

This type of malware is eventually activated to sense if the user context is relevant for the malware. If so, then some other malicious actions are executed. For instance, a malware aiming at spying a very specific industrial system, such as the case of Stuxnet, will remain dormant until the malware hits the target system. Similarly, in a Bring-Your-Own-Device (BYOD) context, malware targeting a specific office building can remain dormant until the device is near a certain location.

Typically, malicious apps are activated when the `BOOT_COMPLETED` event is triggered regardless of the context of the infected device. A recent study on Android malware [Zhou and Jiang, 2012] suggests that the tendency is shifting towards more sophisticated activation triggers so as to better align with the malware incentives and the pursued goals. This results in a variety of more complex activation conditions, such as those shown in Table 7.4.

We instrumented Androrat to activate the RAT component only when the device is in a certain location. We use a mock location near the Bushehr's nuclear plant, simulating a possible behavior for a Stuxnet-like malware. Specifically, the RAT is only activated when the device is near the location: 28.82781° (latitude) and 50.89114° (longitude). Once the RAT is activated, we send the appropriate commands to exfiltrate ambient and call recordings captured through the microphone, the camera, and the camcorder.

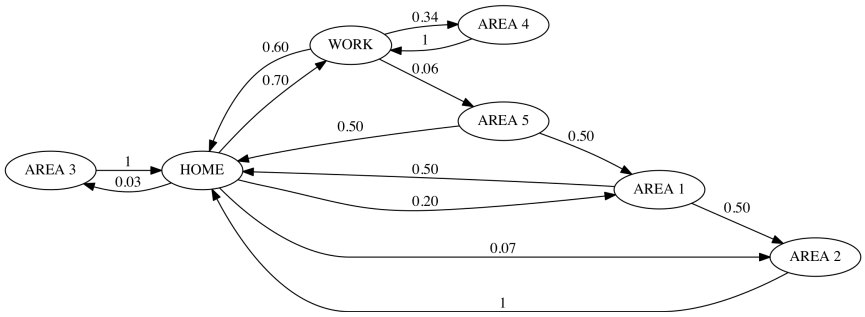


Figure 7.4: Markov chain for the location.

For testing purposes, we built a symbolic model representing the abstract geographic areas of a given user working at Bushehr's plant. Figure 7.4 represents the Markov Model chain for the different areas and the transitions between them. For instance, the model represents a user traveling from HOME (x_H) to WORK (x_W) with a probability of $P(x_H|x_W) = 0.7$.

Given the above model, we then inject testing traces drawn from the chain into the sandbox instrumented with Androrat. The sandbox is configured with a generic RAF aiming at identifying when operations involving personal information occur together with network activity. Results show how the malware is not activated until we start injecting mock locations. A few seconds after the first injection, the behavioral signature collected reported, as expected, both data leakage (`dataleak`) and network activity (`sendnet`).

We next defined an alternative scenario in which an app accesses the user location and sends an SMS to one of his contacts whenever he is leaving a certain region, such as for instance WORK (x_W). To this end, we implement an app and tested it against three users with different contexts and concerns about their privacy. The first user has strict privacy policies and visits very frequently the location x_W. The second user has the same policy as the first one but has never visited such a location.

Finally, the last user visits x_W as well but has a more flexible privacy policy. For the sake of simplicity, we use the same triggering model described in the previous example for user one and three (see Figure 7.4), while the second user has a different Markov chain. Results show that:

- For the first user, the behavioral signature reported data leakage activity (`dataleak`) as well as SMS activity (`sendsms`). As both are in conflict with this user's privacy preferences, this is marked as undesirable behavior.
- In the case of the second user, the model injects locations other than those triggering the grayware component. Consequently, no significant behavioral signature is produced.
- Finally, the events injected for the third user trigger the grayware component, resulting in data leakage and SMS activity. However, as these do not oppose his privacy preferences, no alert is issued.

This example reinforces the view that not only malware activation can be user specific, but that the consequences of such a malware may also be perceived very differently by each user.

7.4.2 Case 2: Anti-analysis Malware

Malware analysis is typically performed in a virtual sandbox rather than in a physical device due to economic and efficiency factors [Suarez-Tangil et al., 2014b]. These sandboxes often have a particular hardware configuration that can be leveraged by malware instances to detect that they are being analyzed and deploy evasion countermeasures [Rogers, 2014], for example by simply not executing the malicious payload

HW feature	Default value
IMEI	000000000000000
IMSI	012345678912345
SIM	012345678912345
Phone Number	1-555-521-PORT (5554)
Model Number	sdk
Network	Android
Battery Status	AC on Charging 50%
IP Address	10.0.2.X

Table 7.5: Default hardware configuration for Android emulator.

if the environment matches a particular configuration. Sandboxes for smartphone platforms have such artifacts. For instance, the IMEI, the phone number, or the IP address are generally configured by default. Furthermore, other hardware features such as the battery level are typically emulated and kept indefinitely at the same status: e.g., AC on and Charging 50%. Table 7.5 summarizes some of these features in most Android emulators along with their default value.

Hardware features such as those described above can be set prior to launching the sandbox. This will prevent basic fingerprinting analysis, for example by setting random values for each execution. However, smarter malware instances might implement more sophisticated approaches, such as waiting for a triggering condition based on a combination of hardware changes. Motivated by this, we modified Androrat to activate the RAT component only after AC is off and the battery status is different from 50%. Once the RAT is activated, we send appropriate commands to exfiltrate some personal information from the device such as SMSs, call history, etc.

In principle, there are as many triggering conditions as combinations of possible hardware events. Although our framework support injection of all possible hardware events via the Android emulator console [Android, 2014], for simplicity we restricted

Status	Health	Present	AC	Capacity
unknown charging discharging not-charging full	unknown good overheat dead overvoltage failure	false true	off on	$0-100\%$

Table 7.6: Different hardware states for power status of the device.

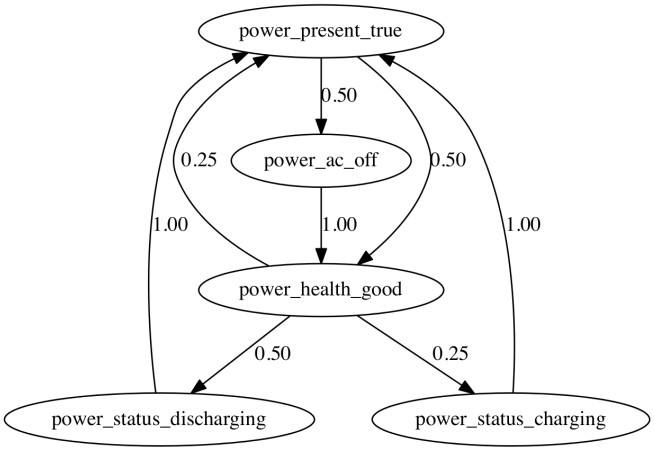


Figure 7.5: Markov chain for the battery status.

our experimentation to the subset of power-related events described in Table 7.6.

Based on the different power states, we built a model of the battery usage extracted from an actual device when used by a real user. The resulting model is shown in Figure 7.5. We then tested Androrat against this model generated using the same RAF configuration used in previous cases. The results show that the behavioral signature not only reported `dataleak` and `sendnet`, but also file activity (`accessedfiles`), thus confirming that the malware activated as it failed to recognize its presence in a sandbox.

7.5 Conclusions

The problem of detecting targeted malware via behavioral analysis requires the ability to reproduce an appropriate set of conditions that will trigger the malicious behavior. Determining those triggering conditions by exhaustively searching through all possible states is an undecidable problem. In this chapter, we have proposed a novel system for mining the behavior of apps in different user-specific usage scenarios and contexts. Our experimental results show that modeling such patterns as Markov chains reduces the complexity of the search space while still offering an effective representation of the usage and context patterns.

Our approach represents a robust building block for thwarting targeted malware, as it allows the analyst to automatically generate patterns of input events to stimulate apps. As the focus of this chapter has been on the design of such a component, we have relied on ad hoc replication and risk assessment components to discuss the quality of our proposal. We are currently extending our system to support: (a) a replication system to automatically generate and test clones of the device under inspection; and (b) a general framework to specify risk assessment functions and analyze behavioral signatures obtained in each clone. Finally, in this chapter we have not discussed the potential privacy implications associated with obtaining user behavioral models. Even if such profiles are just used for testing purposes, they do contain sensitive information and must be handled with caution. This and other related privacy aspects of targeted testing will be tackled in future work.

V

Conclusions, Future Work and References

8

Conclusions

Smart devices equipped with powerful sensing, computing, and networking capabilities have increasingly become the platform of choice—mostly in the form of smartphones and tablets— for many users, outselling the number of PCs worldwide. The rapid development of smartphone technologies and its widespread user acceptance have come hand in hand with a similar increase in the number of malicious software targeting such platforms. This increase is accompanied, in some cases, by sophisticated techniques purposely designed to overcome security architectures and detection mechanisms. This Thesis examines the problem of such *smart* malware and addresses several fundamental issues when automating its analysis in large-scale scenarios.

This Chapter provides the conclusions of this dissertation. We first summarize the main contributions and discuss how they meet the objectives established. Next, we identify and discuss a number of challenging open issues that should be tackled in future work. Finally, we list various results (publications, software, etc.) that have resulted from this Thesis.

8.1 Contributions

We next summarize the contributions made in this work and discuss the main conclusions that arise from them:

1. The comprehensive analysis presented in Chapter 2 on the evolution of malware in smart devices motivates the need for intelligent instruments to automate their analysis. To do so, we have first provided an overview of the security models and protection mechanisms present in current platforms for smart devices. Next, we have proposed a characterization of malware in terms of three key factors: pursued goals and associated behaviors; distribution and infection channels; and privilege acquisition strategies. Our analysis of some representative samples demonstrates that malware is becoming increasingly complex and adaptive, with constantly changing goals and using multiple distribution and infection strategies. We have also provided an analysis of the 20 most significant proposals for detecting and analyzing malware for smart devices proposed between 2010 and 2014. First, we have identified and classified all device features where malware behavior could manifest. This taxonomy has been complemented with additional elements, such as where the monitoring and analysis tasks take place, or the specific detection technique used. Then, we have provided key elements for the design of novel techniques aiming at detecting and analyzing smart malware. Finally, Chapter 3 presents the design and development of a research lab for smart malware analysis and detection. This lab compiles together—and extends—the most cutting-edge open source tools for all static-, dynamic-, and cloud-based analysis. This lab facilitates the automation of smart malware analysis, and we believe it will be extremely

useful to other researchers aiming at automating malware analysis for smart detection.

2. Static analysis is a relatively fast approach to identify malicious software. For this reason, it has been widely used by existing techniques to search for suspicious components. The techniques introduced in Chapter 4 demonstrate that exploiting static features for mining structural patterns in smart malware is extremely efficient. In particular, we have successfully applied a text mining approach to automatically classify smartphone malware samples and analyze families based on the code structures found in them. Our proposal is supported by a statistical analysis of the distribution of such structures over a large dataset of real examples. Our findings point out that the problem bears strong resemblances to some questions arising in automated text classification and other information retrieval tasks. By adapting them to these domains, we have explored the suitability of such techniques to measure similarity among malware samples, and to classify unknown samples into known families. Our results suggest that this technique is fast, scalable, and very accurate. Furthermore, this technique also provides the analyst with a means to analyze the relationships among families, the existence of common ancestors, the prevalence and/or extinction of certain code features, etc. Altogether, this reveals that automated tools are instrumental for analysts to cope with the proliferation and increasing sophistication of malware.

3. Smart malware often relies on obfuscation techniques to avoid detection and to make static analysis harder. Chapter 5 uses dynamic analysis to address this type of malware and introduces a novel technique based on differential fault

analysis to automate its identification in large-scale markets. Our approach demonstrates how fault injection can be used to analyze the behavioral differences between the original app and a number of automatically generated versions of it where a number of modifications (faults) have been carefully injected. Observable differences in terms of activities that appear or vanish in the modified app are used to successfully identify potentially malicious components hidden within an app package. Finally, we show how this approach is a good complement to static analysis tools, more focused on inspecting code components but which could well miss pieces of code hidden in data objects or just obfuscated.

4. Adapting and adopting both static- and dynamic-based analysis tools to battery-powered devices is a challenging problem. Relying on offloaded (i.e., cloud-based) engines has been suggested an alternative for empowering constrained devices with powerful detection capabilities. In Chapter 6, we have discussed the power consumption trade-offs among various strategies for executing anomaly detection components directly on mobile platforms or remotely in the cloud. When researchers are confronted with this dilemma, we have shown that there is actually little choice but to offload everything to the cloud. Reasons for this include the differences between the energy efficiency of computation and communications in current platforms, and also various parameters related to the anomaly detection setting, such as the dataset sizes and the operation frequency.

5. The consumption model previously proposed reveals that the use of detection techniques built in the device is unaffordable. Chapter 7 shows that cloud-based

approaches can be adopted to analyze targeted malware. Current dynamic-based detection strategies are shown to be extremely inefficient when dealing with this type of malware, as analysts must reproduce very specific activation conditions to trigger malicious payloads. First, we position that determining those triggering conditions by exhaustively searching through all possible states is an undecidable problem. To this end, we have proposed a novel system based on cloud cloning for mining the behavior of apps in different user-specific usage scenarios and contexts. We have revealed that using a simple yet powerful stochastic model reduces the complexity of the search space while still offering an effective representation of the usage and context patterns of the targeted device. Finally, we have provided the analyst with a robust system for automatically detecting targeted malware. The main building blocks of this system are: the *evidence generation* subsystem, and the *behavioral modeling and risk assessment* subsystem. The first one extracts usage and context traces from the device and generates the stochastic triggering model. The second block extracts the behavioral signatures from the clone(s) and applies a risk assessment over the evidences collected.

8.2 Open Issues and Future Work

Malware in smart devices still pose many challenges and a number of important issues need to be further studied and addressed with novel solutions. This section identifies some open issues where research is needed.

- **Cooperative security**. In the near future it is very likely that many users will own a *network* of smart devices, including smartphones, smart TVs and other

home appliances, and wearable computing platforms. Such networks could be leveraged to implement cooperative security functions, as a complement to cloud-based and on-platform monitoring and analysis mechanisms. Ideally, several connected devices could cooperate to improve security in a number of ways. For example, resource-intensive tasks can be delegated to devices with a permanent power source to preserve the battery of mobile platforms. Similarly, mutually monitoring schemes could be interesting, where each device monitors the behavior of others to detect compromise.

- **Trusted software**. In the case of current smartphones and tablets, trust on the non-malicious nature of an app is based on two factors: (i) the implicit assumption that the market operator has conducted some security review before making the app available for download; and (ii) the identity of the developer, given by the signature attached to the app, which also provides some evidence of the app's integrity. The first point is not fully reliable, as operators cannot afford to carry out an exhaustive analysis over every submitted app; and, even if they could, there is still some non-negligible probability of sophisticated malware evading detection. As for the identify of the developer and the app's integrity, evidence suggests that most users do not pay much attention to them, or positively ignore them when downloading apps from alternative markets.

  We believe that further efforts to improve trust in software are required. This will be increasingly necessary in the near future, as the number of developers—and, hence, apps—will likely grow very significantly. Reputation systems [Viriyasitavat and Martin, 2012; Zacharia et al., 2000] adapted to this context might offer some added value, in particular by exploiting interactions in large user communities such as, for example, those provided by online social networks

[Govindan and Mohapatra, 2012]. But other mechanisms for building trust could also apply, such as for example remote attestation protocols [Nauman et al., 2010; Saroiu and Wolman, 2010; Viriyasitavat and Martin, 2012] or any other schemes to ensure the authenticity and integrity of software.

- **Malware in other smart devices**. The experience gained from current smartphones suggests that malware will also hit other smart devices as soon as they appear. Evidence in other pervasive technologies already exists. For example, nowadays Radio Frequency Identification (RFID) systems are used in a wide range of applications, such as transport tickets, access control systems, e-passports, e-health applications, etc. The benefits of adopting RFID technology for identification purposes are clear, but its associated security risks need to be addressed. One of them—often underestimated—is malware. The use of Internet-enabled mobile devices as RFID readers makes this sort of attacks potentially more harmful. Most previous works have focused on securing the communication link between the tag and the (mobile) reader. There are, however, some preliminary works [Rieback et al., 2006; Yan et al., 2009] on RFID malware, but further studies and solutions are required. Similarly, Implantable Medical Devices (IMDs) and other medical devices will likely be an attractive target for attackers due to the economic value of the information they can provide [Burleson et al., 2012; Clark and Fu, 2012; Clark et al., 2013].

- **Forensics-based analysis for smart device protection**. Sometimes malicious programs uninstall themselves after achieving their goals. However, analyzing evidences that they leave behind could be used as an input for detecting future propagation using the same infection vector. Identifying such traces is a great

challenge, particularly due to the availability of anti-forensic tools for devices such as smartphones [Distefano et al., 2010]. In this regard, two different approaches might be worth exploring. On the one hand, deleting evidences or attempting to neutralize any source of evidence usually produces fresh new evidences. On the other hand, new paradigms such as the aforementioned replicas in the cloud, allow the creation of novel forensic approaches on the cloud based on virtual introspection.

- **Offloaded security**. Applications are increasingly requiring the user to authorize the transference of personal information to the cloud as part of the normal use of the application. For instance, *WhatsApp* sends the user's address book to establish friendship connections [WhatsApp, 2014]. However, even if the user authorizes such a transference, it does not mean that it will be used for purposes other than those conveyed to the user, such as for example market research. In other cases, users are only informed that some personal information will be sent, but the particulars about what specific items or how it will be used are not given. Identifying misuse of personal information, both on-platform and in the cloud, is a challenging problem that is typically tackled by legal enforcement mechanisms, but technical approaches should be explored. For instance, in the same way that Google App Engine [Google, 2014a] is used to deploy in-the-cloud applications—monitored by Google—, back-end services for smartphones and other smart devices could be moved to a cloud controlled and monitored by a trusted third party. This could make feasible to monitor behavior and enforce security policies in the cloud end of the service, thus complementing other security mechanisms applied in the device.

  Similar privacy-related problems arise in cloud-based monitoring schemes, pri-

marily in those that maintain a virtualized replica of the device to carry out monitoring tasks that are unaffordable to perform directly on the device. Privacy-preserving monitoring systems for this scenario are required, but also more lightweight monitoring and detection mechanisms that can run on platform with an appropriate balance between efficacy and power consumption.

- **Stegomalware**: In the case of smart malware, one commonly observed technique consists of hiding modules containing malicious functionality in places that static analysis tools overlook (e.g., within data objects). More sophisticated hiding techniques, particularly in code, are starting to materialize. These techniques and trends create an additional obstacle to malware analysts, who see their task further complicated and have to ultimately rely on carefully controlled dynamic analysis techniques, such as Alterdroid, to detect the presence of potentially dangerous pieces of code. We believe that smart malware could be using advanced techniques, such as steganography, for concealing their modules within another components of the code. This is specially critical when this components are hidden within distinguishable components (see Alterdroid—Chapter 5).

8.3 Results

All contributions resulting from this Thesis (see Section 1.3 in Chapter 1) have been sent for publication to top ranked peer reviewed journals and international conferences in the Computer Science area. Furthermore, software produced as a result of this Thesis has been sent for copyright protection and made available for *fair use*[1] to the

[1]Permits the use of a copyrighted work for nonprofit or educational purposes.

research community.

This section reports all results published and/or submitted during this PhD, and Table 8.1 presents a summary. The index used for evaluating the journals we aim at this dissertation is the Impact Factor (I.F.) as defined by the Journal Citation Report (JCR) from Thomson Reuters[2]. Similarly, the Computer Research and Education ranking (CORE) from Computer Research & Education[3] is used to evaluate the conferences. These rankings are typically based on the acceptance ratio, number of submissions, citations, and the position of the publication venue with respect to others in the same category.

	Publications		**Indexes**	**Rank**
Journals	*Published*:	3	JCR	Q1
	Submitted:	2	JCR	Q1/2
Conferences	*Published*:	1	CORE	A
Others	*Copyrighted*:	4	–	–
Total		10		

Table 8.1: Summary of the publications of this Thesis and the citation indexes of their corresponding publication venue.

8.3.1 Publications Thesis

We list all publications that arise from this Thesis organized by contribution:

[2] http://thomsonreuters.com/journal-citation-reports/
[3] www.core.edu.au/

P1: "Evolution, Detection and Analysis of Malware for Smart Devices".

- Authors: Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris-Lopez, and Arturo Ribagorda.
- *In: IEEE Comms Surveys & Tutorials*, vol. 16:2, pp. 961-987 (2014).
- I.F. (2012): **4.81**.
- Position in category: 2/132 (**Q1**) in Computer Science.

P2: "Dendroid: A Text Mining Approach to Analyzing and Classifying Code Structures in Android Malware Families".

- Authors: Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris-Lopez, and Jorge Blasco.
- *In: Expert Systems with Applications*, vol. 41, pp. 1104-1117 (2014).
- I.F. (2012): **1.85**.
- Position in category: 56/243 (**Q1**) in Engineering.

P3: "Thwarting Obfuscated Malware via Differential Fault Analysis".

- Authors: Guillermo Suarez-Tangil, Flavio Lombardi, Juan E. Tapiador, and Roberto Di Pietro.
- *In: IEEE Computer*, vol. 47:6, pp. 24-31 (2014).
- I.F. (2012): **1.68**.
- Position in category: 9/50 (**Q1**) in Computer Science.

P4: "Detecting Targeted Smartphone Malware with Behavior-Triggering Stochastic Models".

- Authors: Guillermo Suarez-Tangil, Mauro Conti, Juan E. Tapiador, and Pedro Peris-Lopez.
- To: *European Symposium On Research In Computer Security* (ESORICS), *September 2014*.
- Rank (2013): CORE **A** in Computer Software.

8.3.2 Submitted Publications

We list all works that arise from this Thesis submitted for consideration to be published organized by contribution:

P5: "Alterdroid: Differential Fault Analysis of Obfuscated Malware Behavior".

- Authors: Guillermo Suarez-Tangil, Juan E. Tapiador, Flavio Lombardi, and Roberto Di Pietro.
- To: *IEEE Transactions on Mobile Computing*, *submitted September 2014*.
- I.F. (2012): **2.91**.
- Position in category: 12/135 (**Q1**) in Computer Science.

P6: "Power-aware Anomaly Detection in Smartphones: An Analysis of On-Platform versus Externalized Operation".

- Authors: Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris-Lopez, and Sergio Pastrana.
- To: *Pervasive and Mobile Computing*, *submitted February 2014*.
- I.F. (2012): **1.63**.
- Position in category: 27/172 (**Q1**) in Computer Science.

8.3.3 Copyright Software and Patents

We list the copyright software submissions resulting from this Thesis:

- **Software Registration I:** Alterdroid. This software compiles all source code related to the following contribution: "*Differential fault analysis of obfuscated malware behavior*".

- **Software Registration II:** CloneCloud. "*A stochastic behavioral-triggering model for targeted malware detection*" requires a number of scripts to automatically generate android apps equipped with a number of anomaly detection algorithms.

- **Software Registration III:** CrowdDroid. This software comprises an Android app for monitoring system calls from physical Android smartphones and an Apache server app for collecting such information. This software has been used to evaluate the anomaly detectors described in '*A stochastic behavioral-triggering model for targeted malware detection*".

- **Software Registration IV:** Maldroid Lab. This software compiles all source code related to the following contribution: "*A research lab of malware for smart malware analysis and detection*". Additionally, it compiles software used in:

  - "*A text mining approach for analyzing and classifying malware families*".
  - "*Power-aware anomaly detection in smartphones*".

8.3.4 Research Visits

We finally list the research visits performed during this PhD:

- **Università degli Studi di Padova**: I visited *Dr. Mauro Conti* between September and October 2013. As a result of this visit, we have published the following contribution "Detecting Targeted Smartphone Malware with Behavior-Triggering Stochastic Models" in ESORICS 2014. We are currently working on extending our proposal.

- **Università degli Studi di Roma Tre**: I visited *Dr. Roberto Di Pietro* between September and December 2012. Resulting from this visit, we have published "*Thwarting Obfuscated Malware via Differential Fault Analysis*" in IEEE Computer 2014, and submitted "*Alterdroid: Differential Fault Analysis of Obfuscated Malware Behavior*" to an IEEE Transactions 2014. We are currently working on other proposals.

References

Albert, R. and Barabási, A. L. (2002). Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47–97.

Alll, B. and Tumbleson, C. (Visited May 2014). Dex2jar: Tools to work with android .dex and java .class files. https://code.google.com/p/dex2jar.

Android (Visited May 2014). Android developers. http://developer.android.com/.

Andrus, J., Dall, C., Hof, A. V., Laadan, O., and Nieh, J. (2011). Cells: a virtual mobile smartphone architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 173–187. ACM.

Apple (May 2012). ios security. http://images.apple.com/ipad/business/docs/iOS_Security_Feb14.pdf.

Apple (Visited May 2014). Apple answers fcc questions. http://www.apple.com/hotnews/apple-answers-fcc-questions/.

Apvrille, A. (2011). Cryptography for mobile malware obfuscation. In RSA, editor, *RSA Conference*. Fortinet.

Asokan, N., Davi, L., Dmitrienko, A., Heuser, S., Kostiainen, K., Reshetova, E., and Sadaghi, A.-R. (2013). *Mobile Platform Security*. Morgan & Claypool Publishers, elisa bertino and ravi sandhu edition.

Au, K., Zhou, Y., Huang, Z., Gill, P., and Lie, D. (2011). Short paper: a look at smartphone permission models. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM'11, pages 63–68. ACM.

Auriemma, L. (Visited May 2014). Samsung devices with support for remote controllers. http://aluigi.org/adv/samsux_1-adv.txt.

Backes, M., Gerling, S., Hammer, C., Maffei, M., and Styp-Rekowsky, P. (2012). Appguard —real-time policy enforcement for third-party applications. Technical report, Universitats- und Landesbibliothek, Postfach 151141, 66041 Saarbracken. http://scidok.sulb.uni-saarland.de/volltexte/2012/4902.

Barbera, M. V., Kosta, S., Mei, A., and Stefa, J. (2013). To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *Proc. of IEEE INFOCOM*, volume 2013, pages 1285–1293.

Barrera, D., Kayacik, H. G., van Oorschot, P. C., and Somayaji, A. (2010). A methodology for empirical analysis of permission-based security models and its application to android. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 73–84. ACM.

Batyuk, L., Herpich, M., Camtepe, S., Raddatz, K., Schmidt, A., and Albayrak, S. (2011). Using static analysis for automatic assessment and mitigation of unwanted and malicious activities within android applications. In *6th International Conference on Malicious and Unwanted Software (MALWARE 2011)*, pages 66–72.

Bertrand, A., David, R., Akimov, A., and Junk, P. (Visited May 2014). Remote administration tool for android devices. `https://github.com/DesignativeDave/androrat`.

Bickford, J., O'Hare, R., Baliga, A., Ganapathy, V., and Iftode, L. (2010). Rootkits on smart phones: attacks, implications and opportunities. In *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, HotMobile '10, pages 49–54, New York, NY, USA. ACM.

Blasing, T., Batyuk, L., Schmidt, A., Camtepe, S., and Albayrak, S. (2010). An android application sandbox system for suspicious software detection. In *5th International Conference on Malicious and Unwanted Software*, MALWARE'10, pages 55–62. IEEE.

Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., and Sadeghi, A. (2011a). Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technical report, Technische Universitat Darmstadt.

Bugiel, S., Davi, L., Dmitrienko, A., Heuser, S., Sadeghi, A.-R., and Shastry, B. (2011b). Practical and lightweight domain isolation on android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 51–62, New York, NY, USA. ACM.

Burguera, I., Zurutuza, U., and Nadjm-Tehrani, S. (2011). Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM.

Burleson, W. P., Clark, S. S., Ransford, B., and Fu, K. (2012). Design challenges for secure implantable medical devices. In *Proceedings of the 49th Design Automation Conference*, DAC'12, pages 12–17, New York, NY, USA. ACM.

Cai, L. and Chen, H. (2011). Touchlogger: inferring keystrokes on touch screen from smartphone motion. In *Proceedings of the 6th USENIX conference on Hot topics in security*, HotSec'11, pages 9–9, Berkeley, CA, USA. USENIX Association.

Calvet, J., Fernandez, J. M., and Marion, J.-Y. (2012). Aligot: cryptographic function identification in obfuscated binary programs. In *Proceedings of the 2012 ACM conference on Computer and communications security*, CCS '12, pages 169–182, New York, NY, USA. ACM.

Capilla, R., Ortiz, O., and Hinchey, M. (2014). Context variability for context-aware systems. *Computer*, 47(2):85–87.

Cesare, S. and Xiang, Y. (2010). Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume*, volume 107 of *AusPDC '10*, pages 61–70. Australian Computer Society, Inc.

Chakradeo, S., Reaves, B., Traynor, P., and Enck, W. (2013). Mast: Triage for market-scale mobile malware analysis. In *Proceedings of the Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 13–24, New York, NY, USA. ACM.

Chan, M., Esteve, D., Fourniols, J.-Y., Escriba, C., and Campo, E. (2012). Smart wearable systems: Current status and future challenges. *Artificial Intelligence in Medicine*, 56(3):137–156.

Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM Comput. Surv.*, 41(3):15:1–15:58.

Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A.-R., Shacham, H., and Winandy, M. (2010). Return-oriented programming without returns. In *Proceedings of CCS 2010*, pages 559–72. ACM.

Chibelushi, C., Sharp, B., and Salter, A. (2004). A text mining approach to tracking elements of decision making: a pilot study. In *NLUCS*, pages 51–63. Citeseer.

Chin, E., Felt, A., Greenwood, K., and Wagner, D. (2011). Analyzing inter-application communication in android. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 239–252. ACM.

Chin, E., Felt, A. P., Sekar, V., and Wagner, D. (2012). Measuring user confidence in smartphone security and privacy. In *Symposium on Usable Privacy and Security*, pages 1:1–1:16, Washington. Advancing Science, Serving Society.

Christodorescu, M., Jha, S., Seshia, S., Song, D., and Bryant, R. (2005). Semantics-aware malware detection. In *Security and Privacy, 2005 IEEE Symposium on*, pages 32–46.

Chubb, D. (Visited May 2014). Data privacy of ios 6 release notes. `http://www.product-reviews.net/2012/06/15/ios-6-release-notes-show-heightened-security/`.

Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., and Patti, A. (2011). Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 301–314. ACM.

Clark, S. S. and Fu, K. (2012). Recent results in computer security for medical devices. In *Wireless Mobile Communication and Healthcare*, volume 83 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 111–118. Springer Berlin Heidelberg.

Clark, S. S., Ransford, B., Rahmati, A., Guineau, S., Sorber, J., Xu, W., and Fu, K. (2013). Wattsupdoc: Power side channels to nonintrusively discover untargeted malware

on embedded medical devices. In *Presented as part of the 2013 USENIX Workshop on Health Information Technologies*, Berkeley, CA. USENIX.

Clauset, A., Shalizi, C. R., and Newman, M. E. (2009). Power-law distributions in empirical data. *SIAM review*, 51(4):661–703.

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

Conti, M., Nguyen, V., and Crispo, B. (2011). Crepe: Context-related policy enforcement for android. *Information Security*, 6531:331–345.

Corporation, S. (2013). Internet security threat report. Technical report, Symantex. http://www.symantec.com/content/en/us/enterprise/other_resources/b-istr_main_report_v18_2012_21291018.en-us.pdf.

Crussell, J., Gibler, C., and Chen, H. (2012). Attack of the clones: Detecting cloned applications on android markets. *Computer Security–ESORICS 2012*, pages 37–54.

CuteCircuit (Visited May 2014). T–shirtos: The future is getting closer. http://www.cutecircuit.com/tshirtos-the-future-is-getting-closer/.

Damopoulos, D., Kambourakis, G., and Gritzalis, S. (2011). isam: An iphone stealth airborne malware. In Camenisch, J., Fischer-Hubner, S., Murayama, Y., Portmann, A., and Rieder, C., editors, *Future Challenges in Security and Privacy for Academia and Industry*, volume 354 of *IFIP Advances in Information and Communication Technology*, pages 17–28. Springer Berlin Heidelberg.

Davi, L., Dmitrienko, A., Sadeghi, A.-R., and Winandy, M. (2011a). Privilege escalation attacks on android. In Burmester, M., Tsudik, G., Magliveras, S., and Ilic, I., editors, *Information Security*, volume 6531 of *Lecture Notes in Computer Science*, pages 346–360. Springer Berlin / Heidelberg.

Davi, L., Sadeghi, A.-R., and Winandy, M. (2011b). Ropdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, pages 40–51. ACM.

De Luca, A., Hang, A., Brudy, F., Lindner, C., and Hussmann, H. (2012). Touch me once and i know it's you!: implicit authentication based on touch screen patterns. In *Proceedings of the 2012 ACM annual conference on Human Factors in Computing Systems*, pages 987–996. ACM.

De Meulenaer, G., Gosset, F., Standaert, F.-X., and Pereira, O. (2008). On the energy cost of communication and cryptography in wireless sensor networks. In *Networking and Communications, 2008. WIMOB'08. IEEE International Conference on Wireless and Mobile Computing,*, pages 580–585. IEEE.

Dediu, H. (2012). When will tablets outsell traditional pcs? http://www.asymco.com/2012/03/02/when-will-the-tablet-market-be-larger-than-the-pc-market/.

Dediu, H. (2013). Measuring platform churn. http://www.asymco.com/2013/05/05/platform-churn/.

Dediu, H. (2014b). When will smartphones saturate? http://www.asymco.com/2014/01/07/when-will-smartphones-saturate/.

Dediu, H. (Accessed February 2014a). Systemtap. https://sourceware.org/systemtap/.

Dediu, H., Schmidt, D., and Salle, R. (Visited May 2014). Asymco. http://www.asymco.com/.

Delany, S. J., Buckley, M., and Greene, D. (2012). Sms spam filtering: methods and data. *Expert Systems with Applications*, 39(10):9899–9908.

Desnos, A. (2012). Android: Static analysis using similarity distance. In *45th Hawaii International Conference on System Science*, HICSS'12, pages 5394–5403. IEEE.

Desnos, A. (Visited May 2014). Androguard reverse engineering tool. http://code.google.com/p/androguard/.

Dietz, M., Shekhar, S., Pisetsky, Y., Shu, A., and Wallach, D. S. (2011). Quire: Lightweight provenance for smart phone operating systems. In *Proceedings of the 20th USENIX symposium on security*, page 16. USENIX Association.

Dini, G., Martinelli, F., Saracino, A., and Sgandurra, D. (2012). Madam: a multi-level anomaly detector for android malware. In *Proceedings of the 6th international conference on Mathematical Methods, Models and Architectures for Computer Network Security: computer network security*, MMM-ACNS'12, pages 240–253. Springer-Verlag.

Distefano, A., Me, G., and Pace, F. (2010). Android anti-forensics through a local paradigm. *Digital Investigation*, 7, Supplement:S83–S94.

Dong, M. and Zhong, L. (2011). Self-constructive high-rate system energy modeling for battery-powered mobile systems. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 335–348. ACM.

Dunham, K. (2008). *Mobile malware attacks and defense*. Syngress.

Egele, M., Kruegel, C., Kirda, E., and Vigna, G. (2011). Pios: Detecting privacy leaks in ios applications. In *Proceedings of the Network and Distributed System Security Symposium*, NDSS'11.

Egele, M., Scholte, T., Kirda, E., and Kruegel, C. (2012). A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 44(2):6:1–6:42.

Elish, K. O., Yao, D. D., Ryder, B. G., and Jiang, X. (2013). A static assurance analysis of android applications. Technical report, Virginia Polytechnic Institute and State University.

eMarketer (2014). Smartphone users worldwide will total 1.75 billion in 2014. http://www.emarketer.com/Article/Smartphone-Users-Worldwide-Will-Total-175-Billion-2014/1010536.

Enck, W. (2011). Defending users against smartphone apps: techniques and future directions. In *Proceedings of the 7th international conference on Information Systems Security*, ICISS'11, pages 49–70. Springer-Verlag.

Enck, W., Gilbert, P., Chun, B., Cox, L., Jung, J., McDaniel, P., and Sheth, A. (2010). Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6. USENIX Association.

Enck, W., Octeau, D., McDaniel, P., and Chaudhuri, S. (2011). A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, SEC'11, pages 21–21, Berkeley, CA, USA. USENIX Association.

Enck, W., Ongtang, M., and McDaniel, P. (2009a). On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM.

Enck, W., Ongtang, M., and McDaniel, P. (2009b). Understanding android security. *Security & Privacy, IEEE*, 7(1):50–57.

Erdős, P. and Rényi, A. (1960). On the evolution of random graphs. *Magyar Tud. Akad. Mat. Kutató Int. Közl*, 5:17–61.

Estévez-Tapiador, J. M., Garcia-Teodoro, P., and Díaz-Verdejo, J. E. (2004). Anomaly detection methods in wired networks: a survey and taxonomy. *Computer Communications*, 27(16):1569–1584.

F-Secure (April 2012). Mobile threat report q1 2012. Technical report, F–Secure. "`http://www.f-secure.com/weblog/archives/MobileThreatReport_Q1_2012.pdf`".

F-Secure (Visited May 2014). F–secure mobile threats. `http://www.f-secure.com/en/web/labs_global/mobile-security`.

Falaki, H., Mahajan, R., and Estrin, D. (2011). Systemsens: a tool for monitoring usage in smartphone research deployments. In *Proceedings of the sixth international workshop on MobiArch*, MobiArch '11, pages 25–30. ACM.

Fan, J., Reparaz, O., Rožić, V., and Verbauwhede, I. (2013). Low-energy encryption for medical devices: Security adds an extra design dimension. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 15:1–15:6, New York, NY, USA. ACM.

Fang, Z., Han, W., and Li, Y. (2014). Permission based android security: Issues and countermeasures. *Computers & Security*.

Feizollah, A., Anuar, N. B., Salleh, R., Amalina, F., Ma'arof, R. R., and Shamshirband, S. (2014). A study of machine learning classifiers for anomaly-based mobile botnet detection. *Malaysian Journal of Computer Science*, 26(4).

Felt, A., Wang, H., Moshchuk, A., Hanna, S., and Chin, E. (2011a). Permission re-delegation: Attacks and defenses. In *Proceedings of the 20th USENIX Security Symposium*, pages 1–16. USENIX Association.

Felt, A. P., Chin, E., Hanna, S., Song, D., and Wagner, D. (2011b). Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM.

Felt, A. P., Finifter, M., Chin, E., Hanna, S., and Wagner, D. (2011c). A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 3–14, New York, NY, USA. ACM.

Felt, A. P., Greenwood, K., and Wagner, D. (2011d). The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, WebApps'11, pages 7–7. USENIX Association.

Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., and Wagner, D. (2012). Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA. ACM.

Fenske, J. (2012). Biometrics in new era of mobile access control. *Biometric Technology Today*, 2012(9):9–11.

Fisher, D. H. (1987). Knowledge acquisition via incremental conceptual clustering. *Machine learning*, 2(2):139–172.

Fisk, G., Fisk, M., Papadopoulos, C., and Neil, J. (2003). Eliminating steganography in internet traffic with active wardens. In *Revised Papers from the 5th International Workshop on Information Hiding*, IH '02, pages 18–35, London, UK, UK. Springer-Verlag.

Fleck, D., Tokhtabayev, A., Alarif, A., Stavrou, A., and Nykodym, T. (2013). Pytrigger: A system to trigger & extract user-activated malware behavior. In *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*, pages 92–101. IEEE.

Fleizach, C., Liljenstam, M., Johansson, P., Voelker, G., and Mehes, A. (2007). Can you infect me now?: malware propagation in mobile phone networks. In *Proceedings of the 2007 ACM workshop on Recurring malcode*, pages 61–68. ACM.

Gadia, V. and Rosen, G. (2008). A text-mining approach for classification of genomic fragments. In *Bioinformatics and Biomeidcine Workshops, 2008. BIBMW 2008. IEEE International Conference on*, pages 107–108. IEEE.

Garcia-Teodoro, P., Díaz-Verdejo, J. E., Maciá-Fernández, G., and Vázquez, E. (2009). Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28(1–2):18–28.

Garfinkel, T., Rosenblum, M., et al. (2003). A virtual machine introspection based architecture for intrusion detection. In *Proceedings on Network and Distributed Systems Security Symposium*, volume 3 of *NDSS'03*, pages 191–206.

Gianazza, A., Maggi, F., Fattori, A., Cavallaro, L., and Zanero, S. (2014). Puppetdroid: A user-centric ui exerciser for automatic dynamic analysis of similar android applications. *arXiv preprint arXiv:1402.4826*.

Gilbert, P., Chun, B.-G., Cox, L. P., and Jung, J. (2011). Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services*, MCS '11, pages 21–26, New York, NY, USA. ACM.

Goasduff, L. and Pettey, C. (Visited May 2014). Gartner says worldwide smartphone sales soared in fourth quarter of 2011 with 47 percent growth. `http://www.gartner.com/it/page.jsp?id=1924314`.

Goodin, D. (Visited May 2014). Apple expels serial hacker for publishing iphone exploit. `http://www.theregister.co.uk/2011/11/08/apple_excommunicates_charlie_miller/`.

Google (Visited May 2014a). Google app engine. `www.google.com/enterprise/cloud/appengine`.

Google (Visited May 2014b). Google glass. `http://http://www.google.com/glass/`.

Govindan, K. and Mohapatra, P. (2012). Trust computations and trust dynamics in mobile adhoc networks: A survey. *IEEE Communications Surveys & Tutorials*, 14(2):279–298.

Grace, M., Zhou, Y., Wang, Z., and Jiang, X. (2012a). Systematic detection of capability leaks in stock android smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, NDSS'12.

Grace, M., Zhou, Y., Zhang, Q., Zou, S., and Jiang, X. (2012b). Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM.

Gray, J. (1986). Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12. Los Angeles, CA, USA.

Gudeth, K., Pirretti, M., Hoeper, K., and Buskey, R. (2011). Delivering secure applications on commercial mobile devices: the case for bare metal hypervisors. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 33–38. ACM.

Guido, D. and Arpaia, M. (2012). Mobile exploit intelligence project. `http://www.trailofbits.com/resources/mobile_eip-04-19-2012.pdf`.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The weka data mining software: an update. *ACM SIGKDD Explorations Newsletter*, 11(1):10–18.

Halperin, D., Heydt-Benjamin, T. S., Ransford, B., Clark, S. S., Defend, B., Morgan, W., Fu, K., Kohno, T., and Maisel, W. H. (2008a). Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses. In *Proceedings of the 29th Annual IEEE Symposium on Security and Privacy*, pages 129–142. IEEE.

Halperin, D., Kohno, T., Heydt-Benjamin, T., Fu, K., and Maisel, W. (2008b). Security and privacy for implantable medical devices. *Pervasive Computing, IEEE*, 7(1):30–39.

Hanna, S., Huang, L., Wu, E., Li, S., Chen, C., and Song, D. (2013). Juxtapp: A scalable system for detecting code reuse among android applications. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 7591 of *Lecture Notes in Computer Science*, pages 62–81.

Hao, S., Li, D., Halfond, W., and Govindan, R. (2012). Estimating android applications' cpu energy usage via bytecode profiling. In *Green and Sustainable Software (GREENS), 2012 First International Workshop on*, pages 1–7. IEEE.

Hasan, R., Saxena, N., Haleviz, T., Zawoad, S., and Rinehart, D. (2013). Sensing-enabled channels for hard-to-detect command and control of mobile devices. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 469–480. ACM.

Hastie, T., Tibshirani, R., Friedman, J., and Franklin, J. (2005). The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer*, 27(2):83–85.

Hornyack, P., Han, S., Jung, J., Schechter, S., and Wetherall, D. (2011). These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 639–652. ACM.

Hou, Y.-T., Chang, Y., Chen, T., Laih, C.-S., and Chen, C.-M. (2010). Malicious web content detection by machine learning. *Expert Systems with Applications*, 37(1):55–60.

Huang, H., Zhu, S., Liu, P., and Wu, D. (2013). A framework for evaluating mobile app repackaging detection algorithms. In *Trust and Trustworthy Computing*, pages 169–186. Springer.

Husted, N., Saïdi, H., and Gehani, A. (2011). Smartphone security limitations: conflicting traditions. In *Proceedings of the 2011 Workshop on Governance of Technology, Information, and Policies*, GTIP '11, pages 5–12, New York, NY, USA. ACM.

IIH-uBox (Visited May 2014). Smart pillbox. `http://www.innovatorsinhealth.org/solutions/`.

Jakobsson, M., Shi, E., Golle, P., and Chow, R. (2009). Implicit authentication for mobile devices. In *Proceedings of the 4th USENIX conference on Hot topics in security*, pages 9–9. USENIX Association.

Jensen, C. S., Prasad, M. R., and Møller, A. (2013). Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 67–77. ACM.

Jeon, J., Micinski, K., Vaughan, J., Reddy, N., Zhu, Y., Foster, J., and Millstein, T. (2011). Dr. android and mr. hide: Fine-grained security policies on unmodified android. Technical report, University of Maryland.

Jiang, X. and Zhou, Y. (2013). *Android Malware*. Springer Briefs in Computer Science. Springer.

Jung, W., Kang, C., Yoon, C., Kim, D., and Cha, H. (2012). Devscope: a nonintrusive and online power analysis tool for smartphone hardware components. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 353–362. ACM.

Juniper (2012). 2011 mobile threats report. Technical report, Juniper Networks.

Juniper (2013). 2013 mobile threats report. Technical report, Juniper Networks.

Kalige, E. and Burkey, D. (2012). A case study of eurograbber: How 36 million euros was stolen via malware. Technical report, Versafe.

Karaklajiić, D., Kneževiić, M., and Verbauwhede, I. (2010). Low cost built in self test for public key crypto cores. In *Workshop on Fault Diagnosis and Tolerance in Cryptography*, FDTC'10, pages 97–103.

Ker, A. D., Bas, P., Böhme, R., Cogranne, R., Craver, S., Filler, T., Fridrich, J., and Pevný, T. (2013). Moving steganography and steganalysis from the laboratory into the real world. In *Proceedings of the first ACM workshop on Information hiding and multimedia security*, IH&MMSec '13, pages 45–58, New York, NY, USA. ACM.

Kerckhof, S., Durvaux, F., Hocquet, C., Bol, D., and Standaert, F.-X. (2012). Towards green cryptography: A comparison of lightweight ciphers from the energy viewpoint. In *Cryptographic Hardware and Embedded Systems*, volume 7428 of *Lecture Notes in Computer Science*, pages 390–407. Springer Berlin Heidelberg.

Kim, H., Smith, J., and Shin, K. (2008). Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th international conference on Mobile systems, applications, and services*, MobiSys '08, pages 239–252. ACM.

Knappmeyer, M., Kiani, S. L., Reetz, E. S., Baker, N., and Tonjes, R. (2013). Survey of context provisioning middleware. *IEEE Communications Surveys & Tutorials*, 15(3):1492–1519.

Kosta, S., Aucinas, A., Hui, P., Mortier, R., and Zhang, X. (2012). Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading. In *Proceedings of IEEE International Conference on Computer Communications (INFOCOM)*, pages 945–953. IEEE.

Kostiainen, K., Reshetova, E., Ekberg, J.-E., and Asokan, N. (2011). Old, new, borrowed, blue: a perspective on the evolution of mobile platform security architectures. In *Proceedings of the first ACM conference on Data and application security and privacy*, CODASPY '11, pages 13–24. ACM.

Kraemer, S. and Carayon, P. (2007). Human errors and violations in computer and information security: The viewpoint of network administrators and security specialists. *Applied Ergonomics*, 38(2):143 – 154.

Kramer, S. (2010). Rage against the cage.

Kranz, M., Moller, A., Hammerla, N., Diewald, S., Plotz, T., Olivier, P., and Roalter, L. (2013). The mobile fitness coach: Towards individualized skill assessment using personalized mobile devices. *Pervasive and Mobile Computing*, 9(2):203–215. Special Section: Mobile Interactions with the Real World.

Kumar, K. and Lu, Y.-H. (2010). Cloud computing for mobile users: Can offloading computation save energy? *Computer*, 43(4):51–56.

Kumazawa, T. and Tamai, T. (2011). Counter example-based error localization of behavior models. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, pages 222–236, Berlin, Heidelberg. Springer-Verlag.

Lange, M., Liebergeld, S., Lackorzynski, A., Warg, A., and Peter, M. (2011). L4android: a generic operating system framework for secure smartphones. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, SPSM '11, pages 39–50, New York, NY, USA. ACM.

Langner, R. (2011). Stuxnet: Dissecting a cyberwarfare weapon. *Security & Privacy, IEEE*, 9(3):49–51.

Lantz, P. (Visited May 2014). Android application sandbox. `https://code.google.com/p/droidbox/`.

Larner, S. (2012). Smartphones and tablets in the hospital environment. *British Journal of Healthcare Management*, 18(8):404–405.

Leavitt, N. (2013). Today's mobile security requires a new approach. *IEEE Computer*, 46(11):16–19.

Li, E. and Craver, S. (2011). A square-root law for active wardens. In *Proceedings of the thirteenth ACM multimedia workshop on Multimedia and security*, MM&Sec '11, pages 87–92, New York, NY, USA. ACM.

Li, Q. and Clark, G. (2013). Mobile security: A look ahead. *Security Privacy, IEEE*, 11(1):78–81.

Liang, C.-J. M., Lane, N. D., Brouwers, N., Zhang, L., Karlsson, B., Liu, H., Liu, Y., Tang, J., Shan, X., Chandra, R., et al. (2013). Context virtualizer: A cloud service for automated large-scale mobile app testing under real-world conditions. Technical report, Microsoft.

Liao, S.-H., Chu, P.-H., and Hsiao, P.-Y. (2012). Data mining techniques and applications - a decade review from 2000 to 2011. *Expert Systems with Applications*, 39(12):11303 – 11311.

Lin, Y.-D., Lai, Y.-C., Chen, C.-H., and Tsai, H.-C. (2013). Identifying android malicious repackaged applications by thread-grained system call sequences. *Computers & Security*, 39, Part B(0):340–350.

Linn, C. and Debray, S. (2003). Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM.

Lockheimer, H. (Visited May 2014). Android and security. `http://googlemobile.blogspot.com.es/2012/02/android-and-security.html`.

Lookout (Visited May 2014). Security alert: Hacked websites serve suspicious android apps (notcompatible). `http://goo.gl/yJEgn`.

Lu, L., Li, Z., Wu, Z., Lee, W., and Jiang, G. (2012). Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 229–240. ACM.

Luo, T., Hao, H., Du, W., Wang, Y., and Yin, H. (2011). Attacks on webview in the android system. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 343–352. ACM.

Machiry, A., Tahiliani, R., and Naik, M. (2013). Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA. ACM.

Mahmood, R., Esfahani, N., Kacem, T., Mirzaei, N., Malek, S., and Stavrou, A. (2012). A whitebox approach for automated security testing of android applications on the cloud. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 22–28.

Marquis-Boire, M., Marczak, B., Guarnieri, C., and Scott-Railton, J. (2013). You only click twice: Finfisher,Äôs global proliferation. Research Brief. `https://citizenlab.org/wp-content/uploads/2013/07/15-2013-youonlyclicktwice.pdf`.

McAfee (2011). Threats report:fourth quarter 2010. Technical report, McAfee.

McAfee (2012). Threats report:fourth quarter 2011. Technical report, McAfee.

McAfee (2013). Threats report:fourth quarter 2012. Technical report, McAfee.

Microsoft (2012). Windows phone 8 security overview. Technical report, Microsoft Coorporation. `http://go.microsoft.com/fwlink/?LinkId=266838`.

Milano, D. T. (Visited May 2014). Android view client. `https://github.com/dtmilano/AndroidViewClient`.

Mulliner, C. and Seifert, J.-P. (2010). Rise of the iBots: 0wning a telco network. In *Proceedings of the 5th IEEE International Conference on Malicious and Unwanted Software (Malware)*, pages 71–80, Nancy, France.

Mulliner, C., Vigna, G., Dagon, D., and Lee, W. (2006). Using labeling to prevent cross-service attacks against smart phones. In Buschkes, R. and Laskov, P., editors, *Detection of Intrusions and Malware and Vulnerability Assessment*, volume 4064 of *Lecture Notes in Computer Science*, pages 91–108. Springer Berlin Heidelberg.

Nagata, K., Yamaguchi, S., and Ogawa, H. (2012). A power saving method with consideration of performance in android terminals. In *Ubiquitous Intelligence & Computing and 9th International Conference on Autonomic & Trusted Computing (UIC/ATC), 2012 9th International Conference on*, pages 578–585. IEEE.

NakedSecurity (Visited May 2014). Malicious cloned games attack google android market. `http://nakedsecurity.sophos.com/2011/12/12/malicious-cloned-games-attack-google-android-market/`.

Namboodiri, V. and Ghose, T. (2012). To cloud or not to cloud: A mobile device perspective on energy consumption of applications. In *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pages 1–9.

Natella, R., Cotroneo, D., Duraes, J., and Madeira, H. (2013). On fault representativeness of software fault injection. *Software Engineering, IEEE Transactions on*, 39(1):80–96.

Nauman, M., Khan, S., Zhang, X., and Seifert, J.-P. (2010). Beyond kernel-level integrity measurement: enabling remote attestation for the android platform. In *Trust and Trustworthy Computing*, pages 1–15. Springer.

Newcomb, D. (Visited May 2014). Weblink aims to bridge the nagging smartphone-car disconnect. `http://www.wired.com/autopia/2013/03/weblink-abalta-auto-apps`.

Ni, X., Yang, Z., Bai, X., Champion, A. C., and Xuan, D. (2009). Diffuser: Differentiated user access control on smartphones. In *Mobile Adhoc and Sensor Systems, 2009. MASS'09. IEEE 6th International Conference on*, pages 1012–1017. IEEE.

Nielsen (2012). State of the appnation —a year of change and growth in u.s. smartphones. Technical report, Nielsen.

Nielson, F., Nielson, H., and Hankin, C. (1999). *Principles of Program Analysis*. Springer.

Norris, J. R. (1998). *Markov chains*. Cambridge university press.

Oberreuter, G. and Velásquez, J. D. (2013). Text mining applied to plagiarism detection: The use of words for detecting deviations in the writing style. *Expert Systems with Applications*, 40(9):3756 – 3763.

O'Connor, J. (2006). Blackberry security: Ripe for the picking? Technical report, Symantec.

Ongtang, M., McLaughlin, S., Enck, W., and McDaniel, P. (2009). Semantically rich application-centric security in android. In *Computer Security Applications Conference*, ACSAC'09, pages 340–349.

Panxiaobo (Visited May 2014). Apktool: A tool for reverse engineering android apk files. `https://code.google.com/p/android-apktool/`.

Pathak, A., Hu, Y., and Zhang, M. (2012). Where is the energy spent inside my app?: fine grained energy accounting on smartphones with eprof. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 29–42. ACM.

Pathak, A., Hu, Y., Zhang, M., Bahl, P., and Wang, Y. (2011). Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the sixth conference on Computer systems*, pages 153–168. ACM.

Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C., and Molloy, I. (2012). Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 241–252. ACM.

Porras, P. A., Saidi, H., and Yegneswaran, V. (2010). An analysis of the ikee.b iphone botnet. In Schmidt, A. U., Russello, G., Lioy, A., Prasad, N. R., and Lian, S., editors, *Security and Privacy in Mobile Information and Communication Systems (MobiSec), Second International ICST Conference*, volume 47 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 141–152. Springer.

Portokalidis, G., Homburg, P., Anagnostakis, K., and Bos, H. (2010). Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 347–356.

Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, 1(1):81–106.

Rachuri, K. K., Efstratiou, C., Leontiadis, I., Mascolo, C., and Rentfrow, P. J. (2014). Smartphone sensing offloading for efficiently supporting social sensing applications. *Pervasive and Mobile Computing*, 10, Part A(0):3–21.

Raiu, C. and Emm, D. (2013). Kaspersky security bulletin. Technical report, Kaspersky. `http://media.kaspersky.com/pdf/KSB_2013_EN.pdf`.

Rassameeroj, I. and Tanahashi, Y. (2011). Various approaches in analyzing android applications with its permission-based security models. In *Electro/Information Technology (EIT), 2011 IEEE International Conference on*, pages 1–6.

Rastogi, V., Chen, Y., and Enck, W. (2013a). Appsplayground: automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, CODASPY'13, pages 209–220. ACM.

Rastogi, V., Chen, Y., and Jiang, X. (2013b). Droidchameleon: evaluating android antimalware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium*

on Information, computer and communications security, ASIA CCS '13, pages 329–334, New York, NY, USA. ACM.

Reina, A., Fattori, A., and Cavallaro, L. (2013). A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the 6th European Workshop on System Security*, EUROSEC'13, Prague, Czech Republic.

Rieback, M. R., Simpson, P. N., Crispo, B., and Tanenbaum, A. S. (2006). Rfid malware: Design principles and examples. *Pervasive and mobile computing*, 2(4):405–426.

Rodriguez-Gonzalez, A. Y., Martinez-Trinidad, J. F., Carrasco-Ochoa, J. A., and Ruiz-Shulcloper, J. (2013). Mining frequent patterns and association rules using similarities. *Expert Systems with Applications*, 40(17):6823 – 6836.

Rogers, M. (2014). Dendroid malware can take over your camera, record audio, and sneak into google play. `https://blog.lookout.com/blog/2014/03/06/dendroid/`.

Rohrer, F., Zhang, Y., Chitkushev, L., and Zlateva, T. (2012). Poster: Role based access control for android (rbaca). Technical report, Boston University, MA USA.

Rosen, S., Qian, Z., and Mao, Z. M. (2013). Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 221–232. ACM.

Russello, G., Conti, M., Crispo, B., and Fernandes, E. (2012). Moses: supporting operation modes on smartphones. In *Proceedings of the 17th ACM symposium on Access Control Models and Technologies*, SACMAT '12, pages 3–12, New York, NY, USA. ACM.

Russello, G., Jimenez, A. B., Naderi, H., and van der Mark, W. (2013). Firedroid: Hardening security in almost-stock android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 319–328, New York, NY, USA. ACM.

Ruzgar, E. and Erciyes, K. (2012). Clustering based distributed phylogenetic tree construction. *Expert Systems with Applications*, 39(1):89–98.

Sahin, S., Tolun, M. R., and Hassanpour, R. (2012). Hybrid expert systems: A survey of current approaches and applications. *Expert Systems with Applications*, 39(4):4609–4617.

Salton, G., Wong, A., and Yang, C.-S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620.

Samsung (Visited May 2014). Samsung smart tv. `http://www.samsung.com/us/2012-smart-tv/`.

Saroiu, S. and Wolman, A. (2010). I am a sensor, and i approve this message. In *Proceedings of the 11th Workshop on Mobile Computing Systems & Applications*, HotMobile '10, pages 37–42, New York, NY, USA. ACM.

Schipka, M. (2009). Dollars for downloading. *Network Security*, 2009(1):7–11.

Schmidt, A.-D. (2011). *Detection of Smartphone Malware*. PhD thesis, Universitätsbibliothek.

Schreckling, D., Posegga, J., and Hausknecht, D. (2012). Constroid: Data-Centric Access Control for Android. In *Proceedings of the 27th Symposium on Applied Computing (SAC): Computer Security Track*.

Schrittwieser, S., Katzenbeisser, S., Kieseberg, P., Huber, M., Leithner, M., Mulazzani, M., and Weippl, E. (2013). Covert computation: hiding code in code for obfuscation purposes. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, ASIA CCS '13, pages 529–534, New York, NY, USA. ACM.

Seriot, N. (2010). iphone privacy. *Black Hat DC*, pages 1–30.

Shabtai, A., Fledel, Y., and Elovici, Y. (2010a). Securing android-powered mobile devices using selinux. *Security & Privacy, IEEE*, 8(3):36–44.

Shabtai, A., Fledel, Y., Kanonov, U., Elovici, Y., Dolev, S., and Glezer, C. (2010b). Google android: A comprehensive security assessment. *Security & Privacy, IEEE*, 8(2):35–44.

Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C., and Weiss, Y. (2012). Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38:161–190.

Shabtai, A., Tenenboim-Chekina, L., Mimran, D., Rokach, L., Shapira, B., and Elovici, Y. (2014). Mobile malware detection through analysis of deviations in application network behavior. *Computers & Security*.

Shahzad, F., Akbar, M., Khan, S., and Farooq, M. (2013). Tstructdroid: Realtime malware detection using in-execution dynamic analysis of kernel process control blocks on android. Technical report, National University of Computer & Emerging Sciences, Islamabad, Pakistan.

Shahzad, F., Akbar, M. A., and Farooq, M. (2012). A survey on recent advances in malicious applications analysis and detection techniques for smartphones. Technical report, National University of Computer & Emerging Sciences, Islamabad, Pakistan.

Shi, E., Niu, Y., Jakobsson, M., and Chow, R. (2011). Implicit authentication through learning user behavior. In *Information Security*, pages 99–113. Springer.

Shih, D., Lin, B., Chiang, H., and Shih, M. (2008). Security aspects of mobile phone virus: a critical survey. *Industrial Management & Data Systems*, 108(4):478–494.

Skycure (Visited May 2014). Malicious profiles - the sleeping giant of ios security. `http://blog.skycure.com/2013/03/malicious-profiles-sleeping-giant-of.html`.

Sony (Visited May 2014). Smartwatch. `http://www.sonymobile.com/us/products/accessories/smartwatch/`.

Sophos (Visited May 2014a). First anti–virus software for connected tv. `http://goo.gl/Ww67D`.

Sophos (Visited May 2014b). Sophos endpoint protection. http://www.sophos.com.

Sophos (Visited May 2014c). Sophos mobile security threat report. http://www.sophos.com/en-us/medialibrary/PDFs/other/sophos-mobile-security-threat-report.ashx.

Suarez-Tangil, G., Lombardi, F., Tapiador, J. E., and Pietro, R. D. (2014a). Thwarting obfuscated malware via differential fault analysis. *IEEE Computer*, 47(6):24–31.

Suarez-Tangil, G., Tapiador, J. E., Peris, P., and Ribagorda, A. (2014b). Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys & Tutorials*, 16(2):961–987.

Suarez-Tangil, G., Tapiador, J. E., Peris-Lopez, P., and Blasco, J. (2014c). Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(1):1104–1117.

Symantec (Visited May 2014). Symantec security threats. http://www.symantec.com/security_response/landing/threats.jsp.

Takanen, A., Demott, J. D., and Miller, C. (2008). *Fuzzing for software security testing and quality assurance*. Artech House.

Tan, S. (2005). Neighbor-weighted k-nearest neighbor for unbalanced text corpus. *Expert Systems with Applications*, 28(4):667–671.

Tandel, M. H. and Venkitachalam, V. S. (2013). Cloud computing in smartphone: Is offloading a better-bet? Technical report, Electrical Engineering and Computer Science, Wichita State University.

Thiruvadi, S. and Patel, S. C. (2011). Survey of data-mining techniques used in fraud detection and prevention. *Information Technology Journal*, 10(4):710–716.

Titze, D., Stephanow, P., and Schuette, J. (2013). App-ray: User-driven and fully automated android app security assessment. Technical report, Fraunhofer Institute.

Traynor, P., Lin, M., Ongtang, M., Rao, V., Jaeger, T., McDaniel, P., and La Porta, T. (2009). On cellular botnets: measuring the impact of malicious devices on a cellular network core. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 223–234, New York, NY, USA. ACM.

Verbauwhede, I. (2011). Low budget cryptography to enable wireless security. 4th ACM Conference on Wireless Network Security. WiSec'11.

Verdult, R. and Kooman, F. (2011). Practical attacks on nfc enabled cell phones. In *3rd International Workshop on Near Field Communication*, NFC'2011, pages 77–82.

Viriyasitavat, W. and Martin, A. (2012). A survey of trust in workflows and relevant contexts. *IEEE Communications Surveys & Tutorials*, 14(3):911–940.

Vockley, M. (2012). Safe and secure? healthcare in the cyberworld. *Biomedical instrumentation & technology/Association for the Advancement of Medical Instrumentation*, 46(3):164.

Wang, C., Ren, K., and Wang, J. (2011). Secure and practical outsourcing of linear programming in cloud computing. In *Proceedings of INFOCOMM 2011*, pages 820–828.

Wei, X., Gomez, L., Neamtiu, I., and Faloutsos, M. (2012). Profiledroid: multi-layer profiling of android applications. In *Proceedings of the 18th annual international conference on Mobile computing and networking*, Mobicom '12, pages 137–148. ACM.

WhatsApp (Visited May 2014). Legal info. `http://www.whatsapp.com/legal/?l=en_en`.

Wu, C., Zhou, Y., Patel, K., Liang, Z., and Jiang, X. (2014). Airbag: Boosting smartphone resistance to malware infection. In *Network and Distributed System Security (NDSS)*, NDSS'14, San Diego, CA, USA. Internet Society.

Xiang, C., Binxing, F., Lihua, Y., Xiaoyi, L., and Tianning, Z. (2011). Andbot: towards advanced mobile botnets. In *Proceedings of the 4th USENIX conference on Large-scale exploits and emergent threats*, LEET'11, pages 11–11, Berkeley, CA, USA. USENIX Association.

Xu, Y., Bruns, F., Gonzalez, E., Traboulsi, S., Mott, A., and Bilgic, A. (2010). Performance evaluation of para-virtualization on modern mobile phone platform. In *Proceedings of International Conference on Computer, Electrical, and Systems Science and Engineering*, ICCESSE '10, pages 272–280. Waset.

Yan, L. and Yin, H. (2012). Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX conference on Security symposium*, Security'12, pages 29–29. USENIX Association.

Yan, Q., Li, Y., Li, T., and Deng, R. (2009). A comprehensive study for rfid malwares on mobile devices. In *5th Workshop on RFID Security (RFIDsec 2009 Asia)*.

Yoon, C., Kim, D., Jung, W., Kang, C., and Cha, H. (2012). Appscope: Application energy metering framework for android smartphone using kernel activity monitoring. In *USENIX Annual Technical Conference*, USENIX ATC'12. USENIX Association.

Yu, P., Ma, X., Cao, J., and Lu, J. (2013). Application mobility in pervasive computing: A survey. *Pervasive and Mobile Computing*, 9(1):2–17. Special Section: Pervasive Sustainability.

Zacharia, G., Moukas, A., and Maes, P. (2000). Collaborative reputation mechanisms for electronic marketplaces. *Decision Support Systems*, 29(4):371–388.

Zawoad, S., Hasan, R., and Haque, M. (2013). Poster: Stuxmob: A situational-aware malware for targeted attack on smart mobile devices.

Zhang, L., Tiwana, B., Qian, Z., Wang, Z., Dick, R., Mao, Z., and Yang, L. (2010). Accurate online power estimation and automatic battery behavior based power model generation for smartphones. In *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 105–114. ACM.

Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., and Zou, W. (2012). Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 93–104. ACM.

Zhou, W., Zhou, Y., Grace, M., Jiang, X., and Zou, S. (2013). Fast, scalable detection of piggybacked mobile applications. In *Proceedings of the third ACM conference on Data and application security and privacy*, pages 185–196. ACM.

Zhou, W., Zhou, Y., Jiang, X., and Ning, P. (2012a). Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*, pages 317–326. ACM.

Zhou, Y. and Jiang, X. (2012). Dissecting android malware: Characterization and evolution. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 95–109.

Zhou, Y., Wang, Z., Zhou, W., and Jiang, X. (2012b). Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proc. of the 19th Annual Network and Distributed System Security Symposium*, NDSS'12.

Zonouz, S., Houmansadr, A., Berthier, R., Borisov, N., and Sanders, W. (2013). Secloud: A cloud-based comprehensive and lightweight security solution for smartphones. *Computers & Security*, pages 215–227.

List of Abbreviations

α Slope of a given function, page 187

β Behavioral signature, page 140

$\Delta(\beta_1, \beta_2)$ Differential signature of β_1 and β_2, page 140

γ Average energy consumption, page 187

$\mathbb{A}$ Set of all relevant and observable activities an app can execute, page 138

$\mathbb{C}$ All possible components of an app, page 136

$\mathbb{O}$ Set of all possible signature transformation operators (STOs), page 139

$\mathbf{g}$ Context of the user during the execution of an app, page 138

$\mathcal{B}$ Set of all possible events for all apps, page 210

$\mathcal{D}$ Dataset composed of feature vectors, page 178

$\mathcal{F}_i$ Set of *apps* belonging to the same family i, page 107

$\mathcal{K}$ Set of clusters, page 123

$\mathcal{M}$ Set of malware families, page 108

$\mathcal{P}'$ App resulting after the sequential application of fault injections over $\mathcal{P}$ (see also $\Psi(\mathcal{P})$), page 140

$\mathcal{X}$ Set of all possible contexts, page 211

$\mathsf{dist}(z, v)$ The distance between two vectors, page 120

$\mathsf{len}(\beta)$ The length of signature β, page 139

$\mathsf{sim}(z, v)$ The cosine similarity between two vectors, page 120

$\mathsf{det}(sq)$ Detection model, page 199

$\mathsf{ind}(\psi^{c_i})$ Indistinguishable fault injection operator(ψ^{c_i}), page 141

$\mathtt{CCC}(\mathcal{F}_i)$ Set of common CCs for a family $\mathcal{F}_i$, page 108

$\mathtt{ccf}(c, \mathcal{F}_j)$ Frequency of the feature c in the set $\mathcal{F}_j$, page 115

$\mathtt{CC}(app)$ Set of all different Code Chunks (CCs) found in app *app*, page 107

$\texttt{FCC}(\mathcal{F}_i)$ Set of family Code Chunks (CCs) for a family $\mathcal{F}_i$, page 107

$\texttt{freq}(c, app)$ The number of occurrences of the feature c in app app, page 115

$\texttt{iff}(c, \mathcal{M})$ Inverse family frequency of the feature c with respect to the set $\mathcal{M}$, page 115

$\texttt{I}(c_i, \mathcal{F}_j, \mathcal{M})$ Importance of a feature c_i over the sets $\mathcal{F}_j$ and $\mathcal{M}$, page 115

μ Centroid of a cluster, page 178

ω Frequency of a task, page 189

Π Vector of initial probabilities, page 212

π Base energy consumption factor, page 189

$\Psi(\mathcal{P})$ App resulting after the sequential application of fault injections over $\mathcal{P}$ (see also $\mathcal{P}'$), page 140

$\psi(c)$ Alteration made over a component c (also denoted as ψ^c), page 136

ρ Adjustable detection threshold, page 199

$\tau(c)$ Type of the component c, page 136

C Set of Code Chunks (CCs), page 108

d Vector of features, page 114

$D^{(t)}$ Proximity matrix at level t, page 123

$deg(s_i)$ The degree distribution of a chain is given, page 213

$E(s)$ Energy consumption over per byte nb, page 187

$I_{i,j}$ Importance of the feature i, j (also see $\texttt{I}(c_i, \mathcal{F}_j, \mathcal{M})$), page 115

it Iteration of an algorithm, page 123

$L^{(t)}$ Linkage at level t, page 123

m Term or feature of an object, page 114

$M_{\tau(c)}$ Model that represents a given feature of $\tau(c)$, page 146

$n_{\mathsf{faultApps}}$ Number of fault injections, page 160

nb Number of bytes sent, page 187

$q(t)$ State of a given model at time, page 211

r A cluster, page 123

R_i A given rule i matching a behavior, page 149

$RD(app)$ Redundancy of a app *app*, page 107

s A cluster, page 123

$t_{\text{diffAnalysis}}$ Time of a differential analysis, page 160

t_{exec} Time during which the app is executed, page 160

$t_{\text{genFaultApp}}$ Time to generate a fault injected app, page 160

u Inputs provided by a user during the execution of an app, page 138

v Vector of features, page 115

v_{rs} A feature vector between derived from r and s, page 123

$w_{i,j}$ Measure of the relevance that the i-th term, m_i, has in object d_j, page 114

y Intercept with the y-axes of a given function, page 187

z Vector of features, page 120

A Transition matrix, page 211

app Malware sample represented by a sequence of code structure, page 106

M Markov chain, page 211

N Length of a given property, page 199

O Sequence of observed states, page 212

P An app seen as a collection of components, page 136

t Time taken to perform a task, page 160

W Length of a given feature vector, page 198