University of Minho
School of Engineering

Sara Silva Costa

Security Threats Management in Android Systems

Submitted Thesis at University of Minho for Master Degree in Computers and Industrial Electronic Engineering

Project done under the orientation of

Professor Doutor Henrique Manuel Dinis dos Santos

Professor Doutor Sérgio Adriano Fernandes Lopes

January 2017

Universidade do Minho
Escola de Engenharia

Sara Silva Costa

Gestão de Ameaças de Segurança em Sistemas Android

Dissertação de Mestrado

Mestrado Integrado em Engenharia Eletrónica Industrial e Computadores

Trabalho efetuado sob a orientação do

Professor Doutor Henrique Manuel Dinis dos Santos

Professor Doutor Sérgio Adriano Fernandes Lopes

Janeiro 2017

DECLARAÇÃO

Nome: Sara Silva Costa

Endereço eletrónico: costa.sarasilva@gmail.com Telefone:960452633

Bilhete de Identidade/Cartão do Cidadão: 14090184

Título da dissertação: Gestão de Ameaças de Segurança em Sistemas Android / Security Threats Management in Android Systems

Orientador/a/es:

Professor Doutor Henrique Manuel Dinis dos Santos

Professor Doutor Sérgio Adriano Fernandes Lopes

Ano de conclusão: 2017

Mestrado em Engenharia Eletrónica Industrial e Computadores

Universidade do Minho, ______/______/__________

Assinatura:

“Courage is not the absence of fear but rather the judgement that something is more important than fear.”
— Meg Cabot

ACKNOWLEDGEMENTS

The development of this master's dissertation marks the conclusion of a remarkable chapter, both in my academic and personal life. However, this dissertation was only possible thanks to the support of several people.

First, I want to thank both my supervisors Professor Doutor Henrique Santos and Professor Doutor Sérgio Lopes, for all the help they gave me and for the total availability in the development of this dissertation with advice that I will take with me in the future.

To the University of Minho and to the teachers, who gave me conditions and support in the accomplishment of the dissertation.

Thank my coworkers, who understood my mood changes during the final moments of this dissertation development.

Special thanks to my friends, who have been present in these five years, for all the moments that we have been through, the motivation when things did not go so well, and for the created friendships that will prevail after the end of this chapter of my life.

Finally, special thanks to my parents for everything they have done in my life, where they have always encouraged me to study and to strive for a better future.

Resumo

Com o uso exponencial de telefones para lidar com informações sensíveis, o desenvolvimento de sistemas de intrusão também aumentou. Softwares maliciosos estão constantemente a ser desenvolvidos e as técnicas de intrusão são cada vez mais sofisticadas. Para neutralizar essas intrusões, os sistemas de proteção de segurança precisam constantemente de ser melhorados e atualizados. Sendo o Android um dos sistemas operativos (SO) mais populares, tornou-se também num alvo de desenvolvimento de métodos de intrusão.

As soluções de segurança desenvolvidas monitoram constantemente o sistema em que se encontram e acedendo a um o conjunto definido de parâmetros procuram alterações potencialmente prejudiciais. Um tópico importante ao abordar aplicações mal-intencionadas é a identificação e caracterização do *malware*.

Normalmente, para separar o comportamento normal do sistema do comportamento mal-intencionado, os sistemas de segurança empregam técnicas de *machine learning* ou de *data mining*. No entanto, com a constante evolução das aplicações maliciosas, tais técnicas ainda estão longe de serem capazes de responder completamente às necessidades do mercado.

Esta dissertação teve como objetivo verificar se os padrões de comportamento malicioso são uma forma viável de enfrentar esse desafio. Para responder à pesquisa proposta foram construídos e testados dois modelos de classificação de dados, usando técnicas de *data mining*, e com os dados recolhidos compararam-se os seus desempenhos. Para o desenvolvimento e teste do modelo proposto foi utilizado o software *RapidMiner*, e os dados foram recolhidos através do uso da aplicação *FlowDroid*. Para facilitar a compreensão sobre as potencialidades de segurança da *framework* do Android, realizou-se uma pesquisa sobre a sua arquitetura, estrutura geral e métodos de segurança, incluindo seus mecanismos de defesa e algumas das suas limitações. Além disso, realizou-se um estudo sobre algumas das atuais aplicações existentes para a defesa contra aplicações maliciosas, analisando os seus pontos fortes e fracos.

Palavras-Chave: Segurança Em Dispositivos Moveis, Análise de dados, Sistemas de Deteção de Ameaças, Android, Gestão de Ameaças

ABSTRACT

With the exponential use of mobile phones to handle sensitive information, the intrusion systems development has also increased. Malicious software is constantly being developed and the intrusion techniques are increasingly more sophisticated. Security protection systems trying to counteract these intrusions are constantly being improved and updated. Being Android one of the most popular operating systems, it became an intrusion's methods development target.

Developed security solutions constantly monitor their host system and by accessing a set of defined parameters they try to find potentially harmful changes. An important topic when addressing malicious applications detection is the malware identification and characterization.

Usually, to separate the normal system behavior from the malicious behavior, security systems employ machine learning or data mining techniques. However, with the constant evolution of malicious applications, such techniques are still far from being capable of completely responding to the market needs.

This dissertation aim was to verify if malicious behavior patterns definition is a viable way of addressing this challenge. As part of the proposed research two data mining classification models were built and tested with the collected data, and their performances were compared. the *RapidMiner* software was used for the proposed model development and testing, and data was collected from the *FlowDroid* application.

To facilitate the understanding of the security potential of the Android framework, research was done on the its architecture, overall structure, and security methods, including its protection mechanisms and breaches. It was also done a study on models threats/attacks' description, as well as, on the current existing applications for anti-mobile threats, analyzing their strengths and weaknesses.

KEYWORDS: Mobile Devices Security, Data Analysis, Intrusion Detecting System, Android, Malware Classification

TABLE OF CONTENTS

LIST OF FIGURES

List of Tables

List of Abbreviations

ACG	Activity Call Graph
AIS	Artificial Immune System
ART	Android *RunTime*
AOT	Ahead-Of-Time
API	Application Programing Interface
AUC	Area Under the Curve
CBR	Case-Based Reasoning
CG	Call Graph
CPU	Central Processing Unit
CT-SVM	Clustering Tree based on Support Vector Machines
DDoS	Distributed Denial-of-Service
DGSOT	Dynamically Growing Self-Organizing Tree
DLL	Dynamic-Link Library
DM	Data Mining
DMT	Data Mining Techniques
DNS	Domain Name System
DVM	Dalvik Virtual Machine
DT	Decision Tree
FCG	Function Call Graph
FM	False Malicious
FN	False Normal
HIDS	Host Intrusion Detection Systems
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
ICC	Inter-Component Communication
ICFG	Inter-procedural Control-Flow Graph
ID	Identifier
IDPS	Intrusion Detection and Preventing Systems

IDS	**I**NTRUSION **D**ETECTION **S**YSTEMS
IPC	**I**NTER-**P**ROCESS **C**OMMUNICATION
JIT	**J**UST-**I**N-**T**IME
JVM	**J**AVA **V**IRTUAL **M**ACHINE
KDA	**K**EYSTROKE **D**YNAMICS-BASED **A**UTHENTICATION
MLA	**M**ACHINE **L**EARNING **A**LGORITHMS
NIDS	**N**ETWORK **I**NTRUSION **D**ETECTION **S**YSTEMS
OS	**O**PERATING **S**YSTEM
RAM	**R**ANDOM **A**CCESS **M**EMORY
RBF	**R**ADIAL **B**ASIS **F**UNCTION
ROC	**R**ECEIVER **O**PERATOR **C**HARACTERISTIC
SD	**S**TANDARD **D**EVIATION
SMS	**S**HORT **M**ESSAGE **S**ERVICE
SVM	**S**UPPORT **V**ECTOR **M**ACHINES
TF-IDF	**T**ERM **F**REQUENCY–**I**NVERSE **D**OCUMENT **F**REQUENCY
TM	**T**RUE **M**ALICIOUS
TN	**T**RUE **N**ORMAL
UI	**U**SER **I**NTERFACE
UID	**U**NIQUE **U**SER **I**DENTIFIER
VM	**V**IRTUAL **M**ACHINE

NOMENCLATURE

C	Coefficient
ε	Epsilon
σ	Standard deviation

1. Introduction

1.1 Context and Motivation

Before the widespread use of mobile phones, digital information security was already a great concern for government and enterprises. Criminals, using different types of intrusion methods (i.e. Trojan horses, social engineering, etc.), try to gain access to sensitive information, causing information leakage, which can cause great catastrophes and losses to countries, companies and principally people.

In the past, this concern was directed to the information in computers and servers but now with the exponential use of mobile phones to handle several types of sensitive information that threat has also passed to those devices [1] [2].

In this digital era, the price of information has become of great deal and so an increased effort is made in developing better security methods. However, the resources needed for monitoring and viably respond to large scale security incidents still require an effective solution management. As referred by Cheetancheri et al. (2006), “The challenge is to collect all information from the numerous data sources and to decide on appropriate actions for each reactive component” [3]. Besides the information leakage problem, this type of intrusion can also affect the device correct operation, causing problems such as draining resources (such as battery and RAM) or even corrupted files, compromising its availability and integrity [4].

In parallel with the threats against mobile devices growth, the work on anti-threats mobile applications was also initiated. But, these applications development state is still far from being capable of blocking most existing threats, and the difficulties are even greater when answering to unknown threats. One of the limitations on this field is the lack or limited understanding of the malware developed in mobile devices and the difficulty on getting some studying samples [5].

As stated by Spreitzenbarth et al. (2013), “For understanding the threat to security and privacy it is important for security researchers to analyze malicious software written for these systems.” [6], which combined with the constant evolution of mobile malware implies the constant need for new malware analysis.

1.2 Objectives

This dissertation was developed in the scope of mobile security management solutions and the main purpose is to find malware identification patterns. Thereby this dissertation tries to answer the following research question:

"Is it possible to define measurable behavior patterns in malicious applications?"

However, to find the answer to this question, firstly some other questions need to be solved, namely:

- "What is a behavior pattern?"
- "How to define/classify a behavior pattern?"
- "What defines a malicious application?"
- "How to evaluate a malicious application behavior in a controlled environment?"

1.3 Research Methodology

For this dissertation development, firstly a definition of the work objectives and goals was done, setting the research questions and exploring the existing works and literature on this thematic. The main search keywords used were "host intrusion detection systems", "Android", "taint analysis", "malware classification" and "data mining".

The design science research methodology [7] was applied, specifically to analyze the existing security methods on various Android malware detection solutions. Complementing this research, a study on the Android architecture was made, to understand the system.

Afterwards an Android taint detection application was selected and it was used to preform analysis on various applications.

Finally, an experimental study was performed on the analysis results, where the tests were carefully selected and performed.

1.4 Document Organization

The following chapters 2 and 3 cover a range of background topics to provide a better understanding of the subsequent chapters. Amongst others, Chapter 2 makes an overview to the Android system and chapter 3 gives a short introduction to some security definitions used in the literature.

Chapter 4 presents a state of the art review directed to this dissertation theme. Firstly, some examples are given of computer security applications developed over the years, followed by an overview of security Android applications found relevant to the project, developed in the last 6 years, completed with a comparative review. Lastly, a small review is made on some developed classification malware models.

Chapter 5 presents the experience made in this dissertation, explaining all the steps taken, from the data collection, models building, to their evaluation, and in Chapter 6 are presented the results with their respective analysis.

Lastly, in Chapter 7 is presented this dissertation conclusion, containing an overview of all the accomplishments of the project, its final considerations and limitations, and the suggestions for future work.

2. ANDROID

In this chapter, it is presented an overview of the Android system, centering on the security relevant features of the architecture.

2.1 Introduction to Android

Android is an operating system (OS) designed for smartphones, purchased by Google in 2005, which is binary and source code released as open source software and enabled a rapid grow in the market. The OS has code written in Java, Assembly and C/C++, however the applications are manly written in Java, and sometimes in C/C++ [8].

2.2 Android Structural Overview

The Android structure is divided in five functional layers, as seen in Figure 1, which are the kernel, the native libraries, the Android runtime environment, the framework layer and lastly the applications layer [9] [10]. Note that some authors call the combination of the application framework layer, the native libraries and the Android runtime environment as the middleware layer [11].

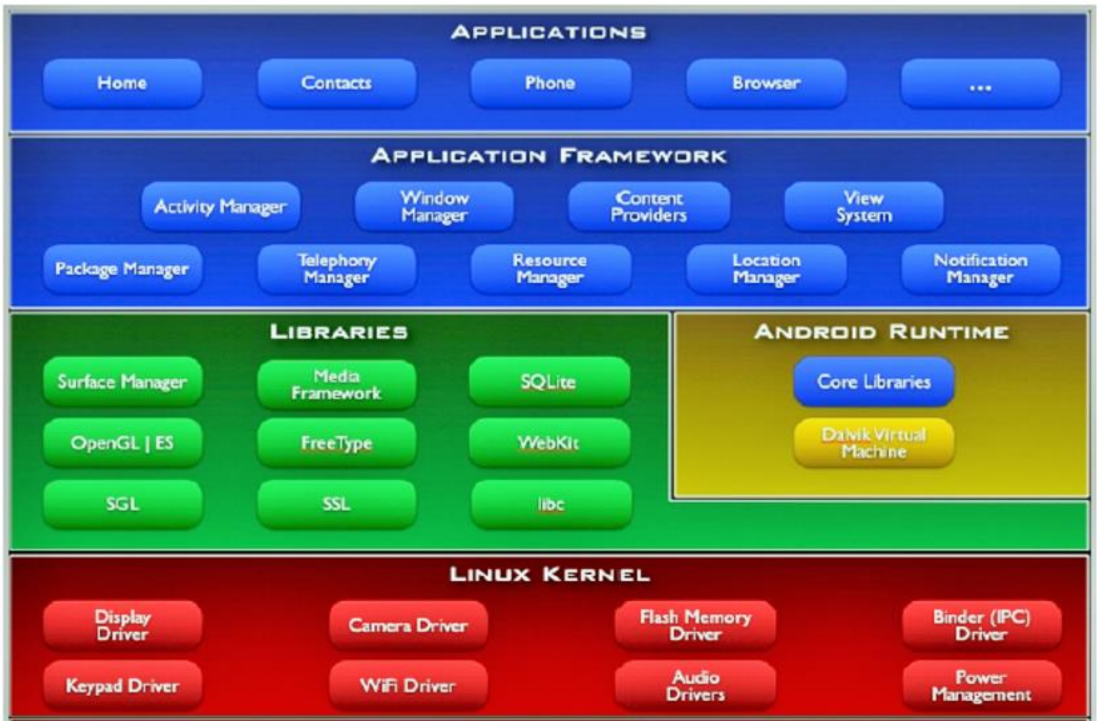


Figure 1 - Structural overview of the Android Software Stack [9]

In the Figure 1 the layers found in blue are programed using Java language, and run in the DVM (Dalvik Virtual Machine) or ART (Android *RunTime*), and the green layers in C/C++ language.

- **Linux kernel:** This basic layer is the Android core system responsible for accessing the underlying hardware, managing device drivers (such as display, camera, Wi-Fi, Binder, etc.), resource access, power management and OS duties.
- **Native libraries:** here can be found various libraries with various functionalities, such as SQLite providing data management, *WebKit* that allows HTML rendering, *Free Tipe*, Secure Socket Layer (SSL) that provides cryptography, the C runtime library (*libe*), *OpenGL* and Media.
- **Android *RunTime* (ART) environment:** this layer consists in an integration between Dalvik VM (DVM) and core libraries functionalities. It replaces just-in-time (JIT) compilation and introduces Ahead-of-time (AOT) compilation, by compiling the applications into persistent native code at installation time, using the tool *dex2oat*, making it directly executable. Although this process makes the installation time longer, the afterwards use is faster since there is no further compilation needs during the application start [12].
  It was firstly introduced with the Android version KitKat (4.4) as an alternative to the DVM and finally replaced it in Lollipop (5.0) version.
- **Applications Framework:** this level allows the developers to communicate with devices built in functions and functionalities, providing an API (application programing interface) in the form of various Java classes.
- **Applications layer:** this layer, seen as the top of the stack, is where the user applications are stored, both the applications provided by each device developer (e.g. browser, dialer phone, etc.) and the ones got from app stores.

2.3 The Dalvik Virtual Machine (Dalvik VM or DVM)

The Dalvik VM is a core component in the Android OS where all applications are executed using its own byte code runtime environment, thereby each application is executed in its own thread with the same priority as core Android applications [9] [13] [14] [15].
Android does not use the standard Java Virtual Machine but uses the DVM which has been tailor made for mobile devices and is an integral part of Android. DVM ensures that applications can run in systems with relatively smaller RAM, slower processors and without swap space in comparison to desktops [16].

The DVM is a register-based bytecode, obtained by translating the Java Virtual Machine (JVM) bytecode using the *dx* tool, producing the Dalvik's executable, namely a *dex* file. To reduce both the machine code generated and the compilation time during the applications runtime, the DMV employs adaptive compilation and indirect threading, by dividing the process into two threads: the main thread and the just-in-time (JIT) compilation thread. Thus, following the illustration shown in Figure 2, firstly in the main thread [i.e. Interpreter thread] the bytecode is executed by the interpreter and when a trace unit of bytecode is detected, i.e. it detects a fragment of hot execution paths, it is send to the second thread [i.e. Compile Thread] and compiled into machine code, thereby only the hot traces are compiled instead of the whole method [17].

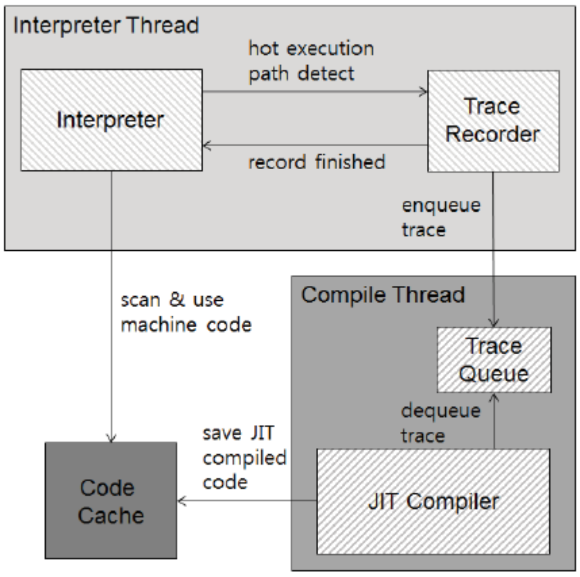


Figure 2 - The Dalvik VM just-in-time compilation process [10]

2.4 Android's Security Model

"Though the OS is immune to normal user usage, but the security flaws can be exploited, as done by the open source community, to get ROOT access (can be used for malicious purposes by crackers) and modify device capabilities" [18]. The core security processes implemented on the Android open-source code are the applications isolation and the permissions system.

2.4.1 Applications Isolation

To prevent malicious applications to manipulate the data of others apps, each application runs in a virtual machine, namely the DVM or ART, providing a sandboxed application execution environment allowing a process isolation of all the applications. Thereby each application has its own low-privileged user space, unique user identifier (UID), only shared between apps signed with the same developer-key [12].

2.4.2 Android Permissions

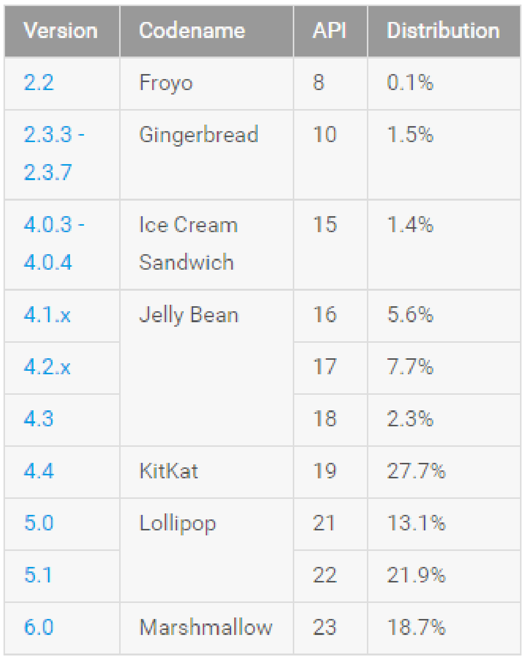
One of the principal security methods implemented in Android is the permission system, which is responsible for controlling which API (Application Programming Interface) calls the applications can access, thus granting access to sensitive data, resources and system interfaces [19] [20].
Depending on the thread level they expose to the system, permissions can be categorized into three levels [21] [22]:

- **Normal permissions:** that have no great impact in the device system nor any cost to the user (e.g. SET_ALARM). They are granted automatically [22] and serve as protection against API calls that could only "annoy the user";
- **Dangerous permissions:** these permissions are granted by the user and can possibly be explored by the application to access an API potentially harmful to the user or device (i.e. privacy leaks, paid SMS services, battery consumption, etc.);
- **Signature/system permissions:** is the most dangerous permission, because it can access and manipulate other applications data. Fortunately, this last permission is only granted to applications signed with the device manufacturer's certificate.
  Peiravian and Zhu [23] divided this last level in two, differentiating if the requesting permission is signed with the same certificate than the application that declared the permission, i.e. both have the same author, or if it is an application of the Android system.

This step allows the user to verify, at installation time, which APIs the application is requesting to access and choose to grant such permissions, gaining better control over privacy issues, thereby as stated by Marforio et al. [24] "users are implicitly lead to believe that by approving the installation of each application independently, based on its declared permissions, they can limit the damage that an application can cause".
However, because of the shared UID, for applications of the same developer they can also share their granted permissions without having to ask them individually [i.e. malicious colluding applications], which means that by installing two or more apps from the same developer, each with their own set of

uncritical permissions, they can collude to get a set of various permissions that individually could be armless but combined may be used to exploit the device' system and perform unauthorized malicious actions [11] [24].

Nonetheless with the Android 6.0 users can now individually decide within each application which permissions they want to accept [12], passing that responsibility interlay to the users. Before if the user didn't want to grant a certain permission, the only option was to abort the app installation, the installation system only finalized the installation by accepting all permissions asked. Nevertheless, since most phone users still use the Android versions between 4.4 and 5.1, as seen in Figure 3, this security issue created by the shared UID cannot be disregarded.



Version	Codename	API	Distribution
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	1.5%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	1.4%
4.1.x	Jelly Bean	16	5.6%
4.2.x		17	7.7%
4.3		18	2.3%
4.4	KitKat	19	27.7%
5.0	Lollipop	21	13.1%
5.1		22	21.9%
6.0	Marshmallow	23	18.7%

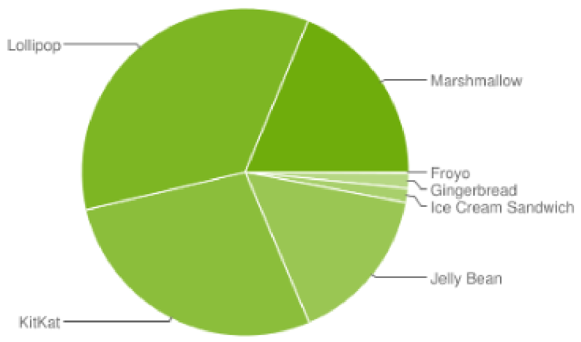


Figure 3 - Relative number of devices running a given version of the Android platform [25]

In addition, as demonstrated in a study by Felt et al. [26], most users ignore this step disregarding its consequences for their device security, or do not understand its information, rendering them unable to take knowledgeable security decisions [11] [27].

Another security breach, presented by Shin et al. (2010) [28], exploits the inexistence of a naming rule restriction or constraint, while declaring a new permission. Without this enforcement, a malicious application can ask for a permission with a normal threat level, which is supposedly harmless to the user, but with the same name of a sensitive permission, associated to a dangerous level, and use it to manipulate the permission specification and verification process, gaining access to sensitive information.

2.5 Inter-Process Communication (IPC)

In the Android architecture, illustrated in Figure 4, the applications are isolated between themselves and from the core system. The IPC, also known as inter-component communication (ICC), allows the exchange of signals or data across multiple processes [29] [30]. This share provides an important tool to development richer applications, keeping a low development burden, and promotes APIs functionalities reuse.

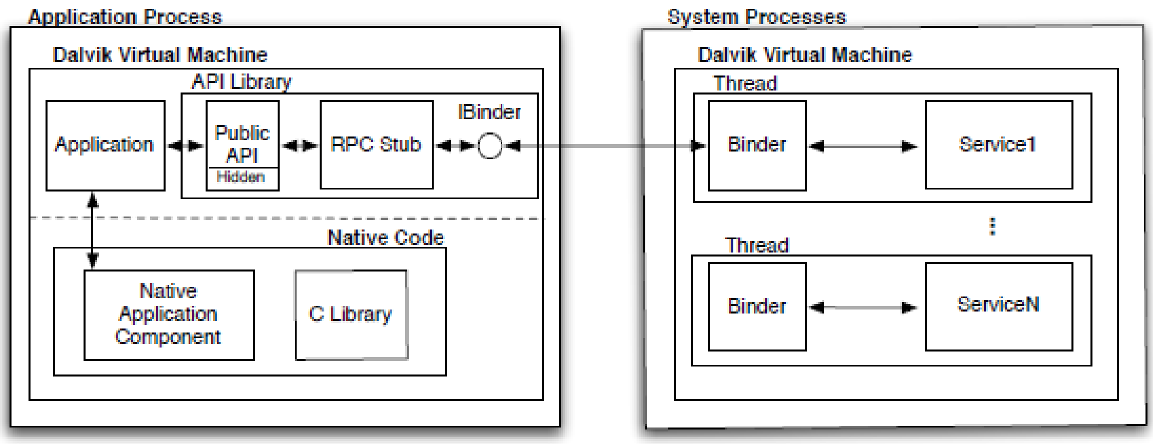


Figure 4 - The architecture of the Android platform [31]

The IPC/ICC process, seen in Figure 5 is Binder-based, i.e., to interact with other applications or with the system processes (e.g. location manager, etc.) an application as to broadcast an intent, i.e., a messaging object used by the applications to request an action or to inform a performed action. Thereafter, the intent is propagated through the binder framework, which handles the intent and forwards it to the destination application or service [16] [32].

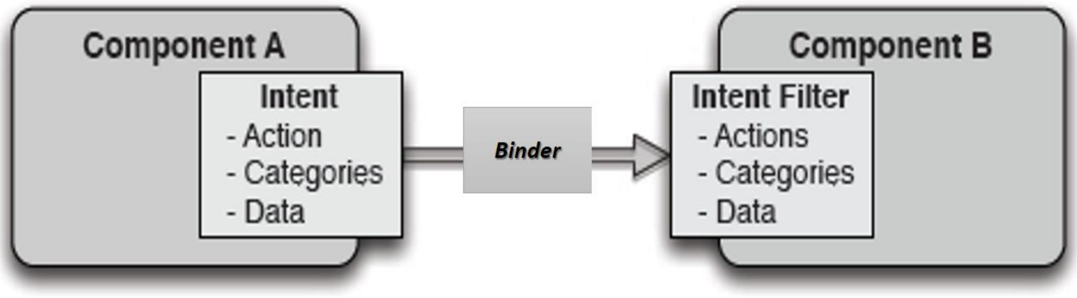


Figure 5 - ICC overview [26]

Unfortunately, this method is supported by complex APIs, which leads to possible vulnerabilities in their implementation, for example, if an intent is left unprotected it can be intercepted by a malicious application or it can leave a sensitive component exposed [33].

Addressing implementation oversighted vulnerabilities is the *ComDroid* application[1], created by Chin et al. [29], that uses the *Dedexer* tool to disassemble the applications *dex* files and, thereafter, examine the IPC model searching for security risks created in its implementation, such as sensitive data loss or corruption or other unexpected behaviors. For the IPC examination *ComDroid* uses static analysis, that will be explained later, and tracks the intents changes from its source to its sink, thus when an intent is send with weak or no permission requirements the system generates a warning. This application facilitates the developer work by allowing him to test its systems for IPC vulnerabilities.

[1] Note that throughout this document sometimes is used “app(s)” as an abbreviation from application(s).

3. Security Background Information

Over the years, the importance of systems security has grown in people's minds. Protection techniques to secure the integrity and confidentiality of the information, such as cryptographic algorithms, digital signatures, authentication biometrics, etc., try to anticipate and avoid possible attacks.

Unfortunately, the development of methods to steal, manipulate or even destroy information has also increased over the years. Usually, the attackers, commonly known as hackers, use different types of vulnerabilities and techniques to gain access to information.

This chapter presents a brief explanation on security concepts contemplated in this document, mainly directed to mobile security.

3.1 Attack

An intrusion can be defined as an ensemble of actions and procedures used to gain access to the systems' information, possibly compromising its confidentiality, integrity and availability [34].

To gain access to the system a cybercriminal usually performs an attack, trying to exploit a vulnerability of the system. If an attack tries to get access to a machine's privileges it is considered a host-based attack, likewise, if it targets the network it is considered network-based (e.g. DDoS attack) [34].

3.2 Mobile threat types

There are several threats that a mobile system can be exposed to, which can be divided into two classes: web-based (i.e. network based) threads, where a cybercriminal relies on the various web features to explore browsers vulnerabilities or tricks an incautious user into providing access to perform an attack, and application based (i.e. host based) threads posed by third party applications (i.e. malicious applications) [6].

Application-based threads can be divided in three main types of threats [35] [36] [37]:

- **Malware:** usually these threats infect the device by installing a malicious app without the user's knowledge or by gaining remote access to the device by exploring its vulnerabilities, and infects the device to steal data, damage the device or even to disturb the user.

These attacks usually work combined with social engineering [9] [10], attacks in which cyber criminals exploit vulnerabilities of human nature, with software and hardware malware in order to accomplish their goals.

- **Personal Spyware:** this type of threat tries to collect the user's sensitive and private data. It differs from the traditional malware because the attacker has physical access to the device and the information collected is send to the person responsible to install the malicious software, which usually isn't the software developer.
- ***Grayware:*** it is less serious than malware since legitimate applications use it to collect user's information and use it for different purposes, such as marketing or user's ambient customizing. This type of information collecting is connected to companies and can be linked to illegal information collection.

3.3 Malware

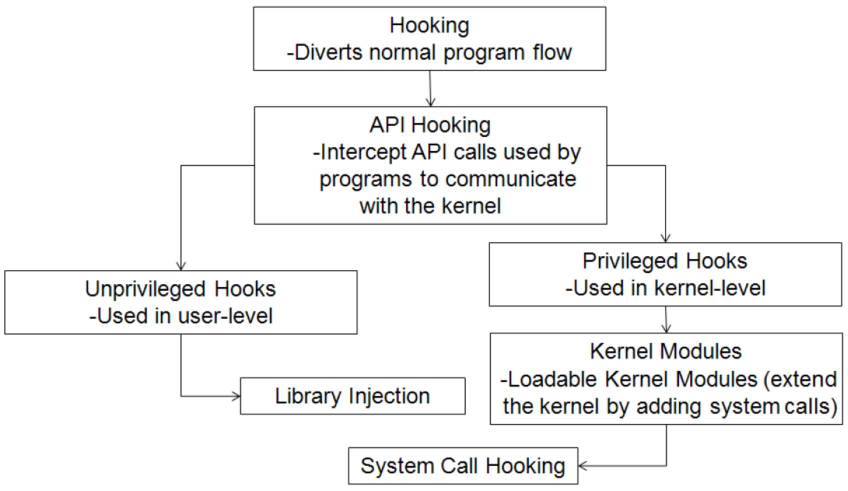
Malware can be spread into devices from various methods, from SMS with links to malicious sites to applications with malicious code. There are several classes of malware attacks that use different approaches to intrude a system. They can be categorized into:

- **Trojan:** a type of malware that infiltrates the device disguised as a normal application and infects systems undetected [38];
- **Virus:** a malware capable of, inside a device, replicate itself infecting other programs [38];
- **Worms:** similar to virus in its replication capabilities, but also capable of exploiting system vulnerabilities by themselves and replicate between devices [39];
- **Bots/Botnet:** these are automated processes used to automatically perform a certain action. Its malicious behavior occurs when used to, for example, send "spam" messages, fill the bandwidth, by performing huge downloads, or making a certain machine or network resource unavailable, by flooding them with requests overloading the system (DDoS attacks) [39] [40] [41];
- **Rootkits:** this category describes infections directed to the device OS, that will afterwards "open" vulnerabilities in the system for other malicious methods; this is usually performed using DLL (dynamic-link library) injection and API hooking techniques [42].

3.3.1 DLL injection and API Hooking

To perform rootkits' attacks developers use techniques such DLL injection and API hooking "that can be used to modify applications without intervening into their source code" [43].

The API hooking technique explores the system by gaining access to the *gDvm* structure, a global structure of type *DvmGlobals* that holds information on a specific Dalvik VM instance [44], and intercepts the system calls between applications and the kernel. The intercepted applications' permissions are used to perform malicious libraries injections (DLL injection) in the address space of a targeted process or adding kernel modules [45]. Presented in Figure 6 is a flowchart of the API hooking algorithm. Note that, depending on the developer intentions, these techniques can also be applied in security enforcement applications.



Hooking
-Diverts normal program flow
API Hooking
-Intercept API calls used by programs to communicate with the kernel
Unprivileged Hooks
-Used in user-level
Privileged Hooks
-Used in kernel-level
Kernel Modules
-Loadable Kernel Modules (extend the kernel by adding system calls)
Library Injection
System Call Hooking

Figure 6 - Overview of API Hooking [46]

3.4 Intrusion Prevention Systems (IPS or IDPS)

There are various methods to detect if a system was infiltrated, however, recently researchers are trying to also generate system responses to the attacks, possible preventing them.

Based on the IDS implementations, prevention based systems use various detecting techniques to determine attacks and furthermore try to stop a detected attack from happening, by blocking it in real-time situations [47] [48].

3.5 Intrusion Detection Systems (IDS)

Intrusion detection systems (IDS) are a collection of the tools, methods, and resources that try to detect system intrusions using different type of approaches and techniques [42] [49] [50].
Detection based approaches uses various detection and data analysis algorithms combined to identify possible malicious activities in devices. Note that there are also hybrid approaches, named intrusion detection and preventing systems (IDPS), which in addition to detecting malicious behaviors also try to stop them [47], as seen in section 3.4.
The wide IDS spectrum is divided into two classification categories: detection methodology and analyzed activity.

3.5.1 Detection methodology

Anomaly detection

The principle of anomaly detection technique is system behavior analyses, searching for abnormalities. Therefore, the expected normal behavior is expressed in the algorithm and an unknown behavior is considered a threat, thus enabling the detection of possibility new threats [51].
Accordingly, if they are defined too strictly, some normal behavior can be considered a threat, originating false positives, and inversely if they are overly flexible some threats may pass undetected, resulting in false negatives [34]. To minimize these results, anomaly detection techniques are applied with self-learning algorithms capable of building models on previous experiences' results. Normally, the data analysis algorithms apply data mining techniques to search for patterns in the data enabling the distinction of behavioral patterns [31].

Misuse detection

Misuse detection is a signature based technique capable of detecting only known threats, using a database model to compare known system traces (i.e. set of attack signatures) left by intrusions [52]. Because, unlike anomaly detection, it does not perform an analysis on the device abnormal behavior, it can only recognize known attacks, rendering it is useless to address unknown intrusions [51] [53].

3.5.2 IDS analysis focus: NIDS and HIDS

IDS can be deployed to analyze events either at the host machine, being known as Host Intrusion Detection System (HIDS), or at the network level, being known as Network Intrusion Detection System (NIDS).

- **Network Intrusion Detection Systems (NIDS):** apply different types of algorithms to analyze huge amounts of network traffic and distinguish those generated by network-based attacks from the originated by normal web operations [34] [54].
- **Host Intrusion Detection Systems (HIDS):** concentrate their efforts in perceiving host-based attacks, which target the device trying to obtain access to its resources or services, applying various processes to monitor the system data and detect abnormal behavior in its applications [34].

  Their main advantage over HIDS is the direct system access, which gives them the ability to access the root cause of attacks [55].

3.5.3 Analysis approach

Nowadays security applications use various approaches to analyze and detect malicious applications that may be infecting the device. Those approaches can be both static and dynamic depending on how they perform the applications analysis.

Dynamic behavior analysis

Security applications based on dynamic taint analysis, also known as behavior based analysis, seem to be one of the most successful rated mechanisms to detect suspicious devices behavior. These security applications focus on examining the operating system by collecting, various device defined parameters and constantly monitoring them to search for notorious differences in their behavior [56] [57].
This method is normally used to look for meaningful differences in the device's characteristics, such as battery consumption, memory load and network utilization. However, "dynamic analysis or behavior-based detection involves running samples in a controlled and isolated environment in order to analyze its execution traces", Zurutuza et al. (2011) [58].

Static behavior analysis

Another procedure to detect applications malicious behavior is to use static behavior analysis, that mainly focuses on detecting sensitive data leakages in applications' components by inspecting other applications source code or binaries looking for suspicious patterns, i.e., the security applications track how the data flow is being handled by other applications, usually at instruction level, searching for leaks of sensitive data [58].

This method differs from the dynamic analysis mainly because it does not perform analysis on the system when the device's applications are working, whereas it can scan the software for malicious patterns without "activating" it, or even without installing it, by sending input data to the application to simulate code execution, as so the number of test runs needed to reach an appropriate code coverage can be large [59]. Another advantage of static analysis over dynamic analysis is the current existence of malware capable of detecting dynamic analysis tools [60].

This method is the most widely used by security companies, however, individually it cannot detect malicious applications that dynamically load code to perform the attacks [61].

3.5.4 User Interface Analysis

Other approach, is to understand if the user interaction with the device is consistent with its applications behavior. For example, if the user is trying to play a game and the game application makes system calls trying to access the user contact information and send it to an unknown entity, using internet and contact information permissions, it may be concluded that there is malicious behavior.

Addressing a different possibility, Hwang et al. (2008) [62] made a study on the use of Keystroke dynamics-based authentication (KDA) for mobile devices to improve the viability of login PINs, which unlike computer logins have small password strings. With this authentication method, the user interface data collected is the user login input time and access to the device is only granted if password input interval times and duration are similar to already registered input rhythms. Although the test results were encouraging it is necessary to extend the input tests, such as studying population and performance measurement types.

3.6 Machine Learning Algorithms and Data Mining

A constraint found in mobile devices is its processing capabilities which, although improved in the latest years, it is still far from comparable to personal computers' processing abilities. Thereby, mobile security solutions need to avoid the devices' overload [63].

To reduce overload on the machine available resources, both on computers and mobile devices, machine learning algorithms (MLA) and/or data mining techniques (DMT) are employed [64]. Usually the same techniques can be applied in both MLA and DMT [65].

Data mining covers a wide spectrum of topics in computer science and statistics. As stated by Hand et al. (2000) "The science of extracting useful information from large data sets or databases is known as data mining."[66]. Some of the tasks in which this science is applied are data characterization and discrimination, association and correlation analysis, classification, clustering, regression, and others [67]. For example, for Android malware classification clustering algorithms are applied in behavior pattern recognition.

These techniques are based on the human ability to learn a mechanism and then self-replicate it to other cases [68]. Summarizing, to employ this method three main steps are needed: 1) feature extraction, responsible for data collecting on the main selected features, 2) classifier construction, in this step the data collected is analyzed, possibly employing various threat detecting methods, and threat assessment is made, and lastly, 3) sequential pattern prediction, where the data behavior is classified as normal behavior or as a threat (malware) [68].

3.6.1 Decision Tree Algorithm

The Decision Tree (DT) algorithm is non-parametric supervised learning method used for classification methods[2], i.e. the algorithm itself grows the number of attributes with the amount of training data, using data patterns it finds to make decisions.

Its goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. It applies a straightforward idea to solve the classification problem [69]. The DT classifier poses a series of carefully crafted questions about the attributes of the test record

[2] It can also be used as a predictability model.

and, according to the obtained calculated answers, it makes a decision about the class label[3] of the data. Note that there several different implementations for this algorithm, that follow different metrics for measuring the data parameters and reaching a decision.

Algorithms for constructing decision trees usually work top-down, i.e. they usually start with all the data in a root element and based on the made decisions they split the data into branches until it reaches the final decision were the data is allocated (called leaves) [70]. A structural example is shown in Figure 7.

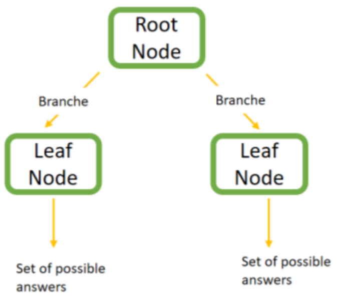


Figure 7 – DT structural example

3.6.2 Support Vector Machine

The Support Vector Machine (SVM) is a highly-sophisticated data mining classification model that given a set of data, with two possible states, tries to divide the hyperplane, where the data is, with decision boundaries, i.e. its objective is to "draw a line" (hyperplane) between both classes of data with the biggest margin possible.

SVM has four well known kernels: linear, polynomial, radial basis function (RBF) and sigmoid, and each represents a different approach to separate the data. Figure 8 represents an example of the division on two classes of data "A" and "B" using a linear approach.

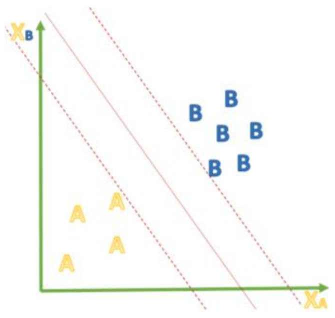


Figure 8 - Example of the division on two classes of data

[3] Class label represents the discrete attribute whose value the model is trying to predict based on the inputted values.

As the name indicates, the linear approach addresses an upfront solution to divide the plane and maximize the margins between a set of two classes of data.

In the example, the classifier used a set of given data points in the form $(\vec{x}_A, y_A), \dots, (\vec{x}_n, y_i)$, where y_i can take the value of both +1 for "A" samples and -1 for "B" samples. The hyperplane can be given as a set of points $\vec{x}$ satisfying:

$$\vec{w} \cdot \vec{x} - b = 0 \qquad \text{(Eq. 1)}$$

In Eq. 1, $\vec{x}$ is a vector to one of the samples, $\vec{w}$ represents a vector perpendicular with the hyperplane drawn with any possible length and b is an unknown value that defines the reach of the unknown sample vector (Note that $\varepsilon = -\mathrm{b}$), where $\frac{b}{\|\vec{w}\|}$ represents the distance between the two margins of the hyperplane.

Note that the introduction of y_i is a mathematical simplification that adds a constraint to the previous equation, Eq. 1, as it can be seen in Eq. 2.

$$y_i(\vec{w} \cdot \vec{x} - b) \geq 1, for\ all\ 1 \leq i \leq n \qquad \text{(Eq. 2)}$$

However not all the data samples can be linearly divided, thereby the *hinge loss* function is introduced to the model, Eq. 3, which takes the value of 0 if the constraint is satisfied, i.e. if $\vec{x}$ is on the correct side of the hyperplane.

$$\max\big(0,1 - y_i(\vec{w} \cdot \vec{x} - b)\big) \qquad \text{(Eq. 3)}$$

If the data is on the wrong side of the margin, the hinge loss function value is proportional to the distance from the margin, which needs to be minimized, Eq. 4.

$$\left[\frac{1}{n}\sum_{i=1}^{n} \max(0,1 - y_i(\vec{w} \cdot \vec{x} - b))\right] + C\|\vec{w}\|^2 \qquad \text{(Eq. 4)}$$

Where the coefficient C represents the level of generalization of the equation, the lower the C value the more generic the model [71].

4. State of the Art

Researchers and companies look for forms of data protection, which implies a constant development and improvement of security management methods. Thereby, several different approaches are used to detect devices' security breaches.

This chapter presents well known examples of security application implementations and some of the most recent developed works found. Firstly, an overview of security works development over the years is done, and afterwards, recent works found relevant for this dissertation are explored.

4.1 Computer security systems

Even before the Android massification there were several IDS proposed for other computing platforms. An earlier work is the *AutoFocus* analysis system, developed by Estan et al. (2003), which focuses on dynamical and multidimensional traffic clusters to automatically generate a traffic behavior report, enabling, in an easier approach, unusual traffic behavior detection. However, the results are limited to periodically display [64].

Liu et al. (2006) proposed a security managing framework while trying to solve some detected problems such as the lack of communication ability between security equipment, the lack of information in security events reports and the unreasonable focus on abnormality detection (i.e. neglecting network equipment or sub-domain host, etc.). Using a CBR module (Case-Based Reasoning) as its main approach to process data correlation, such as user web access, and wavelength diagnoses, studying the different wave frequencies on different network attacks, this framework tries to assimilate and integrate this information [72]. However, there is still the need to develop analysis methods capable of estimating the threat risk level in the obtained information [31].

Zhang and Zulkernine (2006) developed a hybrid system framework, using a combination of misuse and anomaly detection, to detect attacks on the network. The proposed framework applies a misuse detection mechanism that identifies attacks such as patterns or signatures that can represent threats, to classify the network data as normal activity or an attack. It employs the random forest algorithm (i.e. a data mining algorithm), which enables real-time detection. Then the activity classified as normal is reanalyzed using anomaly detection, with low false positive rate, to identify activities with abnormal behavior. In this detection system, the normal behavior is pre-set and the unset behavior is considered unknown. Although this framework has very high intrusion detection rate and it does not require training time, the anomaly detection fails to detect between very similar intrusions, which causes them to fall in the same leaf on the random forest algorithm making them undetectable to the posterior analysis of the anomaly detection system [53].

Khan et al. (2007) presented an algorithm, Clustering Tree based on Support Vector Machines SVM (CT-SVM), capable of reducing false positive and false negative rate results, on real-time, in network level systems. It combines the SVM classification algorithm, a data miming algorithm that reduces training time, with the dynamically growing self-organizing tree (DGSOT) algorithm, a clustering algorithm that enables to find boundary points (i.e. support vectors) used on the SVM training. This combination reduces the SVM training time by inputting only the support vectors which hugely reduces its computation needs, thereby improving network threat detection. Though this method seems promising, the tests realized are still far from granting it full viability and its computing load is yet to be studied on mobile systems [34].

Focusing mostly on HIDS, Cheetancheri et al. (2006) developed a global worm detection algorithm, employed to detect attacks on host-end level. To attain such goal, when the host-end detectors, which are globally connected, detect an anomaly they aggregate their own information and compare it with the conclusions of other (random) chosen host-ends on that anomaly. By using a likelihood ratio table, they deduced if the anomaly is normal behavior or a malicious behavior: if a malicious behavior is detected the information is broadcasted to all host-ends. This method, although capable of detecting worms, has a slow response time making it unable to avoid host infection [3].

Later, inspired in the danger theory of human immune systems, Ou et al. (2011) developed an algorithm employing artificial immune system (AIS) in intrusion detection systems capable of detecting abnormal behavior by analyzing the system calls, considering them as antigens, and evaluating them with three parameters, namely severity, certainty and attack time, to outline the threats [73].

Both Cheetancheri and Ou detection systems, which are host based, have the advantage of addressing the attacks root cause, nonetheless, one of their main problems is the high resource host system consumption, such as processing resources (i.e. memory, CPU).

Analyzing the system calls behavior, Koucham et al. (2015) proposed an algorithm that combines system call argument based clustering with Bayesian classification. For this algorithm, the information of system calls is further explored using its call arguments, contextual information (i.e. identifiers and file models) and domain-level knowledge enabling relevant classification to create improved models for the clusters that reflect differences in the system calls attributes. In this step both the normal and anomalous behaviors are used to create clusters per system call and per process which afterwards provides the classification system, Bayesian classification algorithm, with information on the similarities and differences between the behavior of normal or infected system calls [74].

4.2 Mobile devices security systems

Apart from their portability and OS systems, a big difference between computers and mobile systems is the amount of available resources. In developing security systems for computers, the resources' consumption, such as RAM, is a concern for developers, because a security application if too "heavy" for a system it can render it useless.

Because mobile devices have even less computing resources this is a major problem for its security systems.

Considering this limitation, Addagada (2010) used a neural network to build a user log classifier. In this approach the classifier was pre-trained on a user network activity and a periodic activity security analysis is performed, i.e. the data collected from the device is sent to an analysis server that sends a warning to the user if abnormal behavior is detected. Although this algorithm has good detection rate and low device processing load, the threats are not monitored in real-time, becoming incapable of preventing any prejudicial device activity [75].

4.2.1 Crowdroid

Burguera et al. (2011) proposed an algorithm that extracts the data from various user's devices, using a client directed application called *Crowdroid* responsible for monitoring the system calls done by each application and transfer its accesses to a central server, where the malware is detected based on the amount of system calls made, as seen in Figure 9. That is, analyzing all the system calls happening in each device and comparing them, it considers the benign behavior the most used system calls made by all the devices and as malware behavior the less used system calls.

This algorithm is less resource consuming on the devices and has high detection rate for detecting Trojan horses on the applications. However, if a benign application has more unusual system calls it can generate false positives and the detection accuracy also depends on the number of users connected to de central system, rendering it useless for individual users unconnected to the server [58].

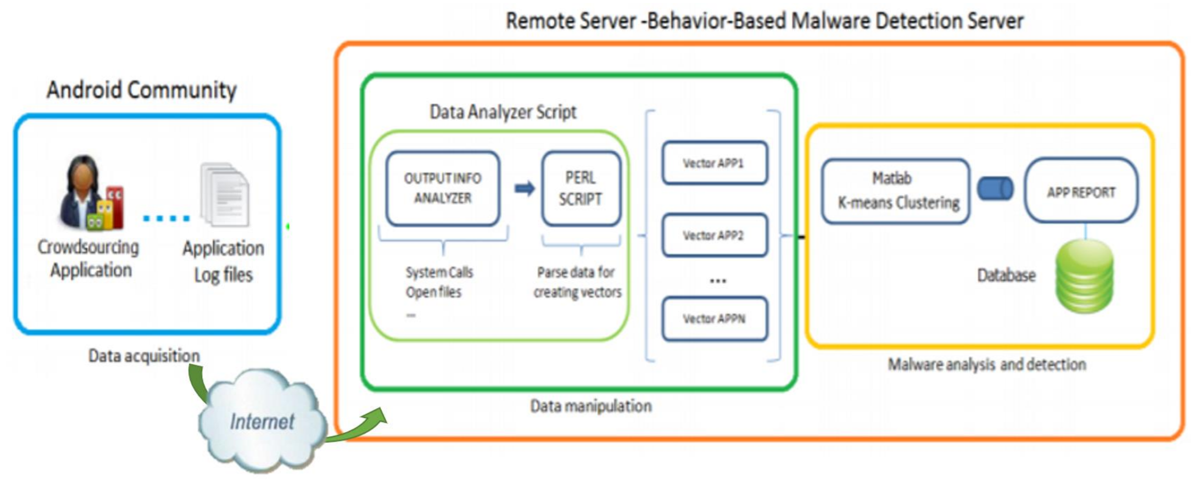


Figure 9 - Crowdroid malware Detection process [57]

Unfortunately, limited systems resources are not the only concern on developing security systems, sometimes the system permissions needed by security enforcement applications are explored creating the possibility of tempering with the implemented IDS, a vulnerability on the application that can possibly render the IDS application useless or even dangerous to the device.

For example, if an attack gains kernel access to the system it can deactivate the IDS or tamper with its configurations rendering it useless. Although virtualization methods were created to address the last problem, there is still need to develop improved solutions [39].

Although not implemented, *Crowdroid*, Khurana et al. (2010) had also already envisioned a possible solution, proposing the implementation of a system process, parallel to an existing HIDS application, that will periodically monitor any changes in the HIDS application architecture or rules and if any unauthorized changes are found it will replace de current HIDS files with backup files [76]..

4.2.2 Andromaly

Shabtai et al. (2012) developed a framework capable of detecting malware by continuously monitoring the mobile device, analyzing a group of pre-selected parameters, such as battery level and CPU consumption, that are fed to a machine learning anomaly detector that searches for abnormal performances.

This application has various advantages such as being light-weighted in processing, running on the device and being modular, making it capable of easily integrating various malware detection techniques.

Although the test results on this approach seemed promising it is still unsuccessful identifying abrupt attacks [77].

4.2.3 Patronus

Sun et al. (2014) realized a study on the API hooking security approach and pointed some vulnerabilities, caused by the implementation method being in the same sandbox as the other applications, specifically the vulnerability of the *insns*. This variable defines a specific address for each method corresponding to its Java method [i.e. target method], that can be modified by the attackers so that it bypasses the policy enforcement, and their capacity of intercepting the native transact method creating their own transact implementation that enables them to sidestep any security policy.

Based on his study, Sun et al. proposed a secure HIPS architecture framework, called *Patronus*, also capable of detecting existing malware. It uses the hooking methodology both in the client and server side, implementing their native transact method directly with the target service, so that its access is unavailable to other applications. In the server side the framework intercepts the Service Manager and redirects the applications requests to the security application which will verify the query realized and, if considered safe, redirect it back to the Service Manager [78].

4.2.4 TaintDroid

A great reference on analysis of third party applications behavior is *TaintDroid*, created by Enck et al. (2014), Yan et al. stated that "(*TaintDroid*) is a special crafted DVM that supports taint analysis of Dalvik instructions across API calls" [79].

TaintDroid assumes that all third-party applications downloaded are untrustworthy, relying on the firmware's integrity: including the virtual machine executed in user space and the loaded native system libraries, and ensures that applications can only use the native libraries from the firmware, denying them (third-party apps) access to libraries downloaded by the untrusted applications.

Unlike other existing solutions, that perform heavyweight whole-system analysis making them unsuitable for real-time analysis, *TaintDroid* defined four focus tracking points for behavior tracking: variable, method, message, and file levels, creating a full system information-flow tracking device capable of analyzing various sources of sensitive data and reducing the runtime performance overhead.

To track the system at multiple granularities, following Figure 10 representation, the system firstly creates a "decoy-flag", by using a trusted application that handles sensitive information such as the location provider, to generate tainted information (1), that is then stored together as one taint tag. When the taint interface invokes a native method (2) that interacts with the Dalvik VM interpreter to store the tag in the virtual taint map and afterwards propagate it (3). Afterwards, the taint tag is send through the modified Binder (4) to the Binder of the kernel (5), then through the Dalvik VM virtual map (6 and 7), until it reaches the untrusted application. Using the created tag, when the untrusted application invokes a library tagged as a taint sink (8), libraries with sensitive permissions such as network that may allow the app to send information to the web, the library retrieves the taint tag and knows that the information is sensitive (9) reporting the event [80].

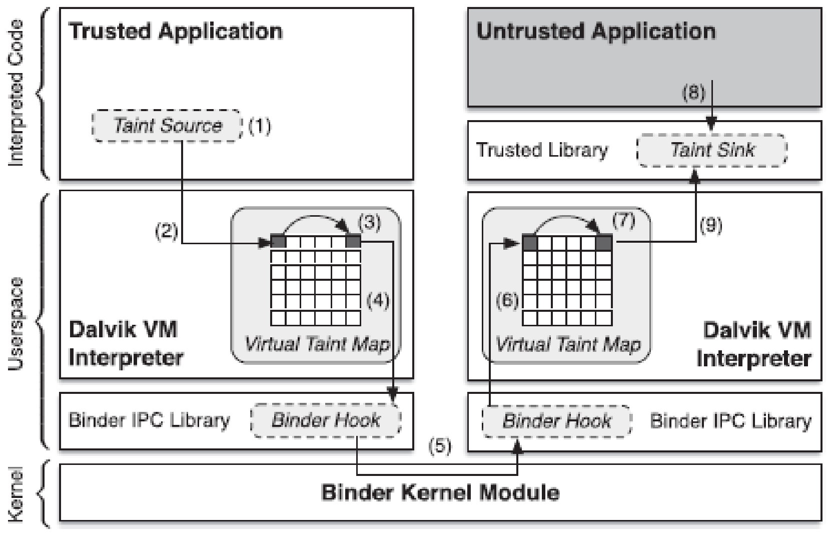


Figure 10 - TaintDroid architecture [80]

This application was then used as a craft design for new security applications development. For example, *DroidBox* [81], which is an open source project that uses *TaintDroid* to build an application sandbox environment, on Android, to dynamically taint analyze the machine code level, making it able of detecting malicious behavior in Java components, native components or components based on both. Although, these approaches have positive detection rates they still face some shortcomings, for example, on applications installation time the *TaintDroid* system can only detect malware if paired with a dynamic testing approach [60]. Another problem to address is the existence of malicious applications that are only triggered by the user interaction with the tainted application. Thereby, a static approach, that only analyses the applications code before their execution, will not be able of detect possible information leakages [82].

4.2.5 SmartDroid

SmartDroid is a prototype system, developed by Zheng et al. (2012), that combines the static analysis with dynamic analysis to detect application malicious behavior. It does so by comparing the system calls data, generated by the application, with the user interface (UI) triggered conditions, which in case of behaviors mismatch is considered suspicious behavior. Figure 11 shows the system architecture.

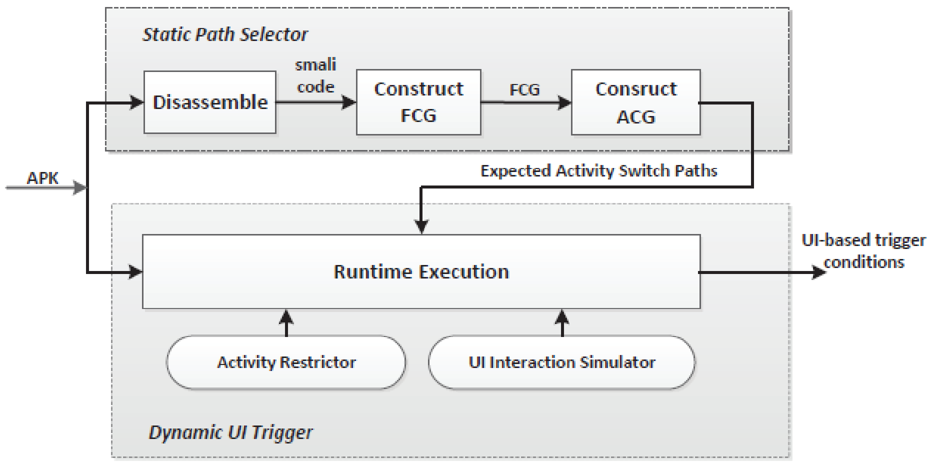


Figure 11 - SmartDroid architecture [82]

The static analysis is used, to firstly, produce an expected activity switch path, that will guide the dynamic analysis on its interaction with each activity.

Thereunto, in the static analysis the application intents to construct a function call graph (FCG), used to determine all function call paths to sensitive APIs, and an activity call graph (ACG), used to define the activity related to the sensitive source functions. Combining this data an expected activity switch path is created and then used in the dynamic analysis stage, together with the UI interaction simulator, to determine which UI elements can trigger this behavior on the application. With this information, the application automatically analyses the system applications searching for malicious behaviors. Although, the presented results seem positive for simple indirect trigger conditions, based on UI, as stated by the authors the system still cannot reveal some complex indirect conditions between the device user behavior with the system calls generated [82].

4.2.6 AsDroid

Identically to *SmartDroid*, *AsDroid* (Anti-Stealth Droid), proposed by Huang et al. (2014), also compares the user's interactions with the applications behavior to detect suspicious behaviors. However, unlike taint analysis techniques that only allow the detection of malicious apps that perform information leakage, *AsDroid* searches for stealthy behavior in malicious apps even if there is no information leakage.

AsDroid root approach is slightly different, the Android APIs are classified into different groups, each associated to an intent type [i.e. *SendSms*, *PhoneCall*, *HttpAccess*, etc.], these intents are then used by the two components that make up this technique: the static program analysis and the UI analysis.

In the static program analysis, reachability analysis is performed on control flow graph (CFG), "used in computer science to represent all of the paths a program may traverse during execution of the program" [83], and on call graph (CG), represents the relationships between the program subroutines [84], and the intents are propagated from the API call sites (i.e. behavior of interest) to top level functions with associated UI. On the UI analysis component, the UI artifacts, corresponding to the button and its residence dialog, are identified and its text is extracted, analyzing it to identify a set of keywords.

Subsequently to both analysis, the data collected, from the top-level functions and UI artifacts, is searched for compatibilities and analyzed for correlation dependencies between intents, if there is no relation between the intents and a mismatched behavior occurs it is confirmed to be found a stealthy behavior [85].

4.2.7 FlowDroid

FlowDroid is an application, created by Bodden et al. (2013), that performs static taint analysis for Android applications searching for information leakages [60].

Its method is to search the application's bytecode and its configuration files, such as layout XML file and executable *dex* files, searching for lifecycle and callback methods and calls to sources, sinks and entry points in the analyzed application. Note that another project of the authors previously ascertains the sources and sinks, namely *SuSi*, a machine learning approach that identifies sources and sinks from the Android source code [86].

As stated by *FlowDroid*'s authors "Sources are calls into resource methods returning non-constant values into the application code. (...) Sinks are calls into resource methods accepting at least one non-constant data value from the application code as parameter, if and only if a new value is written or an existing one is overwritten on the resource" [86].

Then it generates a main method, from the collected information, that is used to generate a call graph (CG), which combines pre-extracted Android callback information to find unknown callbacks, that is fed to an inter-procedural control-flow graph (ICFG), an exploded super-graph based on flow functions associated with the program statements. Taint analysis is performed by starting in the sources detected and tracking taints by traversing the ICFG, i.e. after finding a possible taint source it follows the code until it reaches an app process kill or it finds a sink, which means that between that source and sink a taint if found.

This approach technically resembles *LeakMiner* approach, as seen in Figure 12, a technique that performs static analysis on applications on market site before there are distributed to users [87], however their distinguishing point is the context-sensitive analysis performed by *FlowDroid*, which states that it causes the system success rate to improve [88].

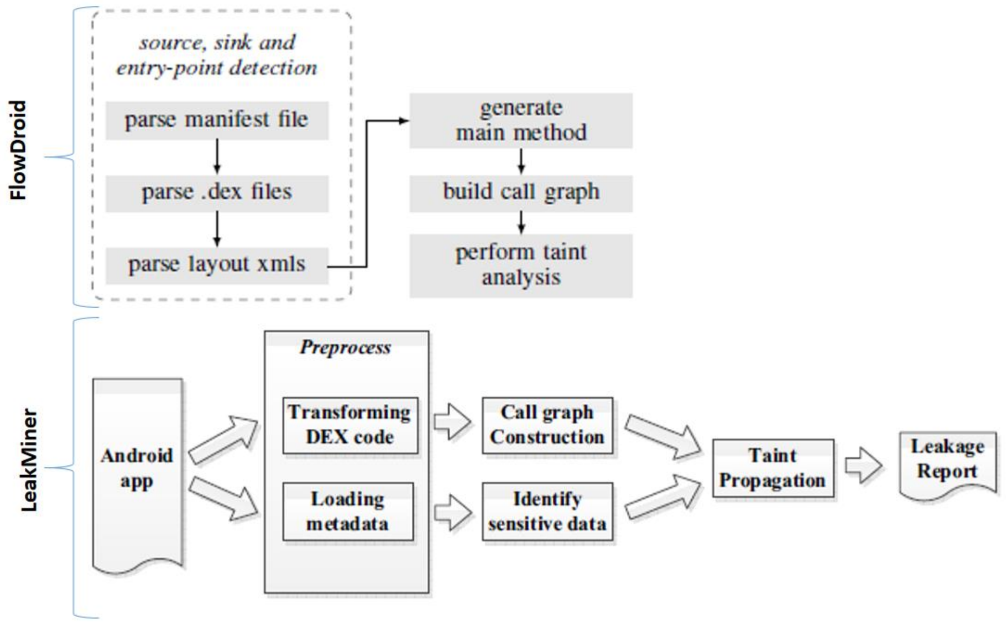


Figure 12 – Comparative overview of FlowDroid and LeakMiner architectures [60] [87]

4.2.8 Applications review resume

There are several works developed on improving the security management methods, for both mobile and computer platforms. Whereas this dissertation focus is on mobile devices, to perform a comparative review of the works analyzed in this dissertation, resumed in Table 1 only mobile system' works are listed.

Table 1 - Resume of mobile security applications

System Name / Author	IDS analysis focus	Detection methodology	Analysis type	Data mining technique	Resumed characteristics

n.s.[4] / Addagada	NIDS	Anomaly	static	Naïve Bayes Neural Network;
				SVM;
					•good detection rate;
					•low device processing load;
					•threats monetarization not on-time;

Crowdroid / Burguera et al.	HIDS	Anomaly	n.s.	K-means clustering;	•monitors the system calls; •high detection rate Trojan horses; •can generate false positives; •useless offline; •threats monetarization not on-time;

Andromaly / Shabtai et al.	HIDS	Anomaly	dynamic	K-means;
				Logistic regression;
				Histogram;
				DT;
				Bayesian Networks;
				Naïve Bayes;
					•searching for abnormal performances;
					•light-weighted;
					•runes on the device;
					•modular;
					• unsuccessful to identify abrupt attacks;

Patronus / Sun et al.	HIDS/ NIDS	Anomaly	dynamic	*ptrace*-based analysis system;	•solves an API hooking security approach vulnerability;
TaintDroid / Enck et al.	HIDS	Anomaly	static	n.s.	•taint analysis application; •denies apps access to libraries downloaded by the untrusted applications;
SmartDroid / Zheng et al.	HIDS	Anomaly	static and dynamic	Fixed-point algorithm	•compares the system calls data with UI triggered conditions; •searches for suspicious behavior; •system fails to reveal some complex indirect conditions;
AsDroid / Huang et al.	HIDS	Anomaly	static	n.s.	•detects stealthy behavior;
FlowDroid / Bodden et al.	HIDS	Misuse / Anomaly	static	n.s.	•static taint analysis; •detects possible information leakages;

From the resume, most security applications employ more than one type of security method to detect malicious applications in mobile systems, improving a lot both in their detection rate and system overall

[4] "n.s." stands for not stated.

consumption. They also detect different types of malware, using different parameters in feature detection, and different data mining (or machine learning) techniques to classify the applications behavior.

However, because of the broad scope of possible malware and to address this dissertation problem, it was decided to focus on data leakage malware, i.e. flows of data between sources and sinks with malicious purposes such as information stealing.

4.3 Malware Classification Models

An important subject when addressing malicious systems is the malware classification analysis. When dealing with malicious detection data, for computerized systems, the size of inputted data to be analyzed is impractical for a hand-made analysis. Thereby, data mining techniques, or machine learning, are often used to transform the initial data into humanly readable information or in relevant information for the security algorithms.

An applied advantage of these analysis is the possible use of its output results to generate malicious signatures, used in the development of future security systems.

For example, Jang et al. (2011) developed a system capable of analyzing malware that provides a perception in the differences and similarities between and within malware data sets, named *BitShred*. The *BitSherd* application uses Boolean featured malicious variables combined with Jaccard similarity metric, that calculates the percentage of common features, and hashing implementation to make comparisons in the malware data and find correspondences [89].

On the NIDS systems approach, Perdisci et al. (2012) developed a clustering model to aggregate HTTP-based malware into similar clusters based on its behavior patterns [90]. Thomas and Mohaised (2014), also on the NIDS systems approach, proposed a DNS traffic data analysis model based on clustering models that looks for patterns on the Botnet domains names used for the command and control of Botnets [91].

5. METHODOLOGY

5.1 Problem Contextualization

Developing security applications systems capable of detecting, and possibly disable, malware applications faces a major challenge with the constant evolution of malicious software. When facing this challenge, understanding the malware contained in those applications is paramount. Recognizing the importance and complexity of this theme, this dissertation addresses the problem by answering the research question posed: "*Is it possible to define measurable behavior patterns in malicious applications?*".

5.1.1 Malicious Applications

An advantage of the Android OS is its open source availability, which provides developers a wide scope of features when developing their own applications, and grants access to the Android applications market. Unfortunately, this easy access also facilitates the development and spread of malicious applications by criminals.

As explained in Chapter 3 there are several methods and attack types. Usually application based attacks are performed by malicious applications, a third-party application that can perform different types of actions harmful for both the device or its owner, for example installing other apps without user consent, subscribing to premium services and change or leak data [92].

5.1.2 Behavior Patterns

A behavior pattern can be defined as a repetition of an action or a set of dominant behaviors, for example, for Beaucamps et al. (2010) "Behavior patterns are defined after observing malicious execution traces and extracting basic sequences likely to be part of a malicious behavior. These patterns often define usual interactions of the program with the system or the network. Once extracted, a sequence either defines a new behavior pattern or extends an existing pattern" [93].

In this dissertation scenario, the variables in this study are the flows between the sources and sinks existing in an application, thereby their repetitions in different applications are considered behavior patterns.

To explore patterns in collected data, two classification models are used: a Decision Tree model and a Support Vector Machine model, which employ the patterns in the data to make predictions for new data.

5.2 Experimental Setup

As aforementioned, this dissertation aim is to find if it is possible to define measurable behaviors patterns in malicious applications. Since the data obtained constitutes a great deal of "raw" material it is impossible to analyze it manually, thereby data mining (DM) techniques are used, i.e. algorithms created to extract various types of information (e.g. knowledge extraction, data archaeology, data dreading, data/pattern analysis, etc.) [67], resorting to the *RapidMiner*[5] software [94] to apply them. Table 2 presents a summary of all the *RapidMiner* operators used.

Table 2 - Resume of all the workflow operators used to build all the models

Operator [95]	**Used functionality**
Retrieve	This operator is used to access the data repositories.
Set Role	Used to change the role of the attributes.
Filter examples	Allows to separate the samples from a repository with unwanted data, in this context it was used to remove the unknown data samples.
Generate TFIDF	Generates a numeric statistic, by filtering the database, which reflects the importance of each word in the database.
Nominal to Numerical	Converts (bi)nominal data into representative numeric values.
Apply Model	Used to apply a trained model on a database.
Performance	Makes a modal performance estimation.
Cross Validation	Estimates the statistical performance a given learning operator will have in a real performance.
Decision Tree	Applies the Decision Tree algorithm.
SVM	Applies the SVM algorithm.
Optimize Parameter (Grid)	Finds the optimal values for a set of selected parameters.

[5] Note that all the operators used in data preparing, application and evaluation of the models were from *RapidMiner.*

5.3 Data Collection

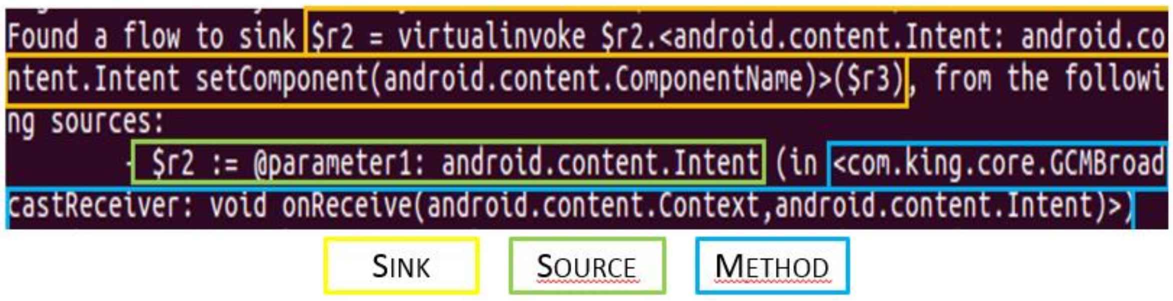
When defining behavior patterns any system needs generated measurement data. In Chapter 4 an overview of some security applications developed for mobile systems is made. From the applications studied, considering their code availability and documentation, the *FlowDroid* application was chosen to perform the analysis on third-party applications and its outputted data was collected for the experiment done. As stated (in section 4.2.7) the application performs leakage detection on applications, outputting information of the sources and sinks found, which composed the data used for this dissertation experiment.

All the used applications were randomly chosen. The normal applications were obtained from the "*Google Play*" store [96] (e.g. "Facebook", "Messenger", "Candy Crush", "VLC", "Photo Editor Pro", etc.) and the malicious applications were gotten from "*Contagium*" malware blog [97] (e.g. "AndroidTrojAt", "Skype", "Brain Test Ghost Push", "Sms worker", etc.).

As seen, a requirement of the *FlowDroid* program is a source and sink document (i.e. SourcesAndSinks.txt), obtained from *SuSi* application [86]. The sample size was decided based on the number of sources and sinks in the *SourcesAndSinks.txt* document, which had 147 sources and 165 sinks, resulting in the decision of using 24 application samples, from which 13 samples were normal applications (approximately 520 flows) and 11 were malicious applications samples (approximately 385 Flows).

The total number of flows collected is 905 flows, however its repartition is not exact because all the flows obtained in the malicious applications analysis were considered as malicious. However, depending on the application purposes and implementation, it could contain normal flows.

Firstly, using *FlowDroid* several analyses where performed, both on normal applications and malicious applications. Figure 13 presents an example of the *FlowDroid* output data, where the sink, source and method are highlighted.

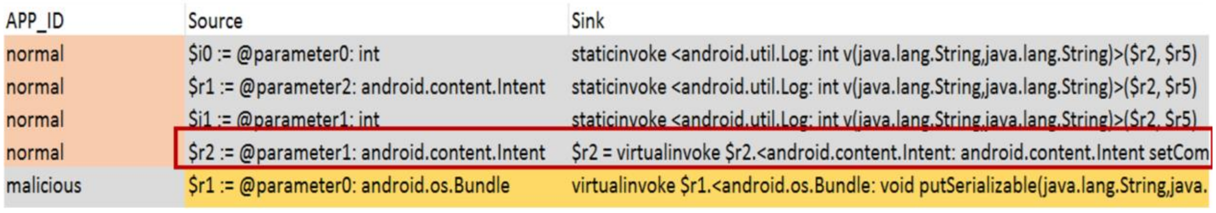


```
Found a flow to sink $r2 = virtualinvoke $r2.<android.content.Intent: android.co
ntent.Intent setComponent(android.content.ComponentName)>($r3), from the followi
ng sources:
        - $r2 := @parameter1: android.content.Intent (in <com.king.core.GCMBroad
castReceiver: void onReceive(android.content.Context,android.content.Intent)>)
```

Figure 13 - Example of FlowDroid output data

After performing the analysis, the data was collected to an *excel* file, example shown in Figure 14. It should be noted that the method data, outputted by the *FlowDroid* application, was not considered in this analysis because it corresponds to specific features of each application, being irrelevant at this point of the analysis.

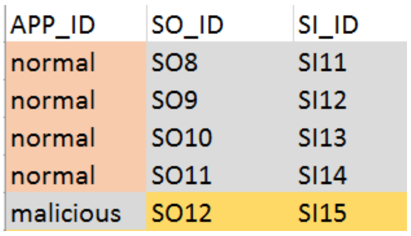
As it can be seen in Figure 14, every line represents a flow, i.e. in the tested application information is read from the device (source) and is sent "outside" (sink). The "APP_ID" column represents the knowledge on the flow, if it is identified as "normal" the data flow is known as from a normal application, if it is identified as "malicious" the data flow is from a malicious application. The highlighted flow seen in Figure 14 represents the data collected from the example presented in Figure 13.



APP_ID	Source	Sink
normal	$i0 := @parameter0: int	staticinvoke <android.util.Log: int v(java.lang.String,java.lang.String)>($r2, $r5)
normal	$r1 := @parameter2: android.content.Intent	staticinvoke <android.util.Log: int v(java.lang.String,java.lang.String)>($r2, $r5)
normal	$i1 := @parameter1: int	staticinvoke <android.util.Log: int v(java.lang.String,java.lang.String)>($r2, $r5)
normal	$r2 := @parameter1: android.content.Intent	$r2 = virtualinvoke $r2.<android.content.Intent: android.content.Intent setCom
malicious	$r1 := @parameter0: android.os.Bundle	virtualinvoke $r1.<android.os.Bundle: void putSerializable(java.lang.String,java.

Figure 14 - FlowDroid output data in excel file example

For a pre-analysis on the data collected, a handmade abstraction of the data was initially performed, where a unique "ID" was given for each different app (source and sink). Taking for example, the data seen in Figure 14, the transformed equivalent data is shown in Figure 15.



APP_ID	SO_ID	SI_ID
normal	SO8	SI11
normal	SO9	SI12
normal	SO10	SI13
normal	SO11	SI14
malicious	SO12	SI15

Figure 15 – Figure 12 data ID transformation example

5.3.1 Data Preparation

Before building the analysis models, data preparation was performed, as shown in Figure 16. First, the database was filtered (1), in order to remove the unknown flows, i.e. those with "APP_ID" unknown.

To avoid losing significant data information a TF-IDF (term frequency–inverse document frequency) filtering was performed (2), to infer about the importance of certain word represented in the database, where each word is given a certain value of importance (term frequency value). This operation was performed by enabling the “calculate_term_frequencies” parameter, of the “Generate TFIDF” operator, that indicates that the term frequency values would be generated by *RapidMiner*.

The aim of this model is to find relations (patterns) in the inputted data to classify it, thereby it is required to normalize the given data (4), so that all the data was in the same range, however the data obtained are strings. To enable the normalization, the use of numeric data is imperative[6], for therefore a transformation from nominal data type[7] to numeric data was done (3).

Finally, the resulted data was saved, in a *RapidMiner* database, and later used in the experiment.

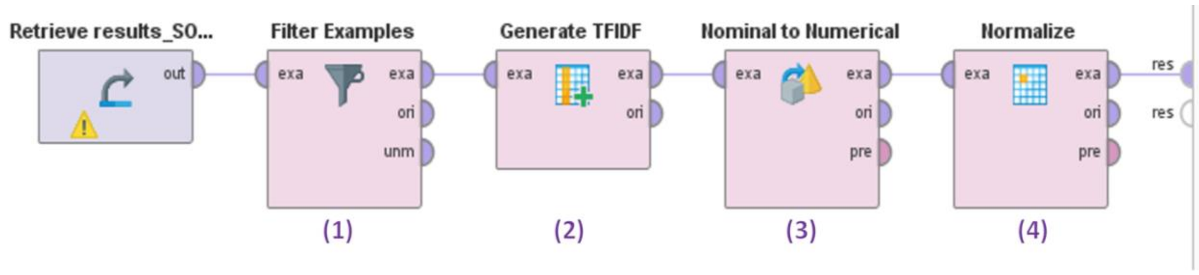
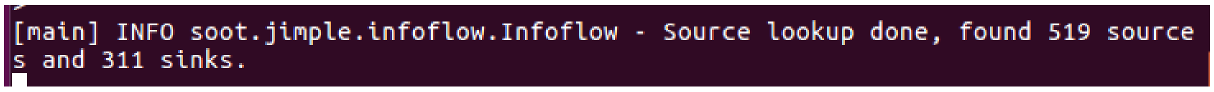


Figure 16 - Workflow of the data preparation

5.4 Data Pre-analysis

To get a better understanding of the collected data, a pre-analysis was made before building the data mining models. This analysis was carried out by directly observing the data and graphic representation. As seen in Figure 17, a sink can have more than one source, the output information has sources (519) and sinks (311).

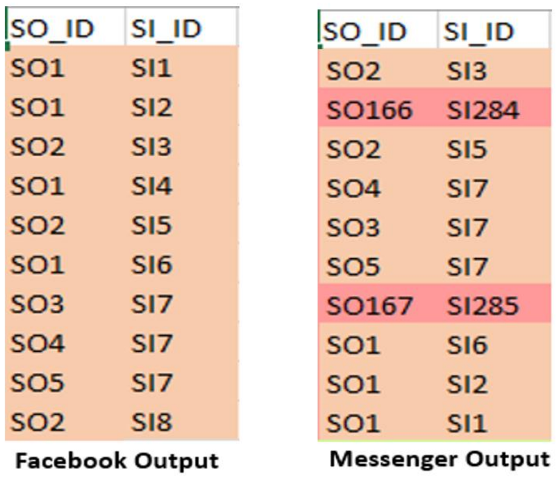


```
[main] INFO soot.jimple.infoflow.Infoflow - Source lookup done, found 519 source
s and 311 sinks.
```

Figure 17 - FlowDroid number of sources and sink found in a app analysis

By directly analyzing the *excel* file transformed into IDs, example shown in Figure 15, it was observed that similar applications usually have more sources and sinks in common. The “Facebook” and “Messenger” applications, for example, which are from the same author and have most of their flows in common (vide Figure 18).

[6] The normalization performed by *RapidMiner* uses numerical values [95].
[7] Note that the same operator is used both for nominal and binominal data.



SO_ID	SI_ID
SO1	SI1
SO1	SI2
SO2	SI3
SO1	SI4
SO2	SI5
SO1	SI6
SO3	SI7
SO4	SI7
SO5	SI7
SO2	SI8

Facebook Output

SO_ID	SI_ID
SO2	SI3
SO166	SI284
SO2	SI5
SO4	SI7
SO3	SI7
SO5	SI7
SO167	SI285
SO1	SI6
SO1	SI2
SO1	SI1

Messenger Output

Figure 18 - Similar applications flows comparison

Afterwards, a direct output of the normalized data, transformed into IDs, was made. From the Figure 19, where the blue data ("0") are normal applications flows and the red data ("1") are the malicious applications flows.

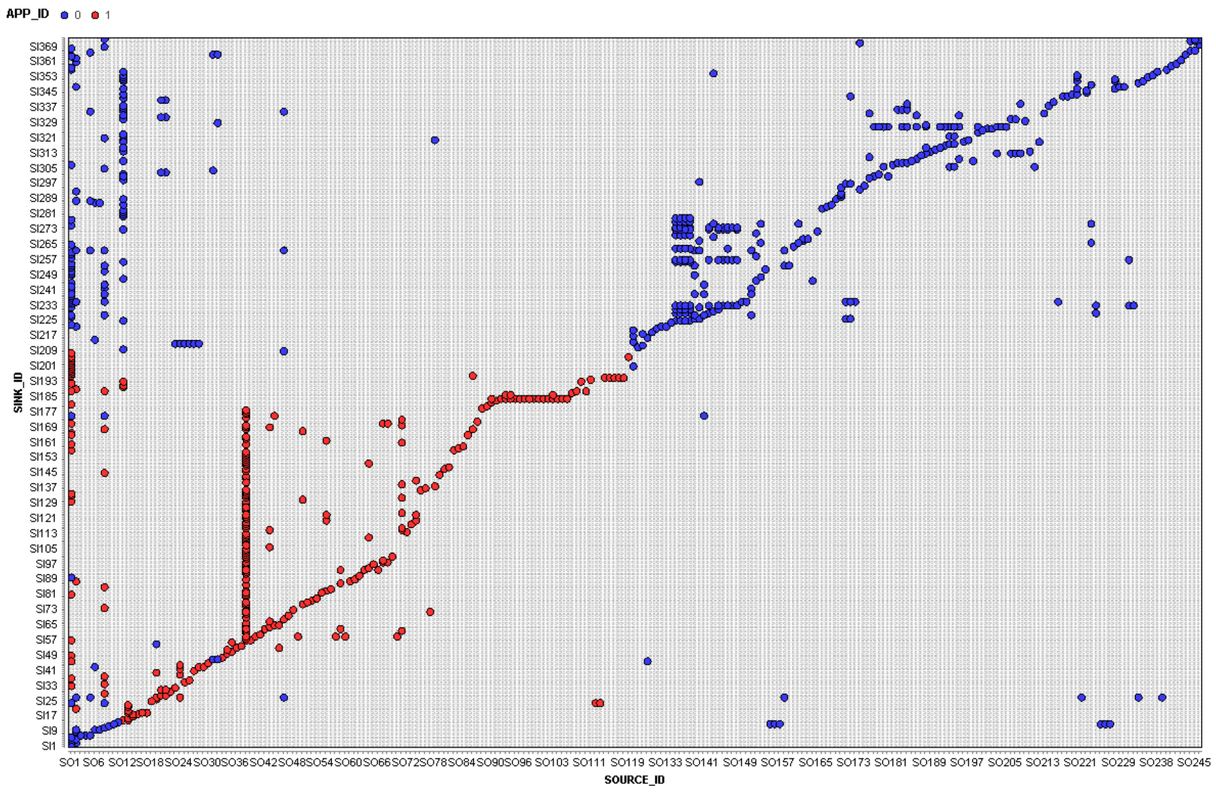


Figure 19 - Graphic output of the transformed IDs, gotten from RapidMiner results

5.5 Data Analysis Models

To begin the data testing the classification techniques chosen were Decision Tree (DT) and Support Vector Machine (SVM). This decision was made based on the data collected and on the most frequently analysis models found during the literature review.

Also, some unsuccessful preliminary tests were done using clustering models. However, considering this dissertation's goal, to prove that it is possible to define malicious applications behavior patterns and not to find the best pattern recognition model, it was decided to use the classification models mentioned.

5.5.1 Decision Tree Model

The Decision Tree (DT) classification model, similarly to an inverted tree architecture, is applied by starting with all its data in the root and consecutively splitting the data into smaller portions and allocating it into the nodes, until it reaches a classification decision in the leaves. An advantage of this technique is its ability to build models from datasets with both numerical and categorical data [70] [69]. Figure 20 stands for a representation of the DT model workflow. Note that the databases of training data ("dados_treino") and classification data ("classificação_treino") were obtained previously, and saved, by the output from the data preparation step, showed in section 5.3.1.

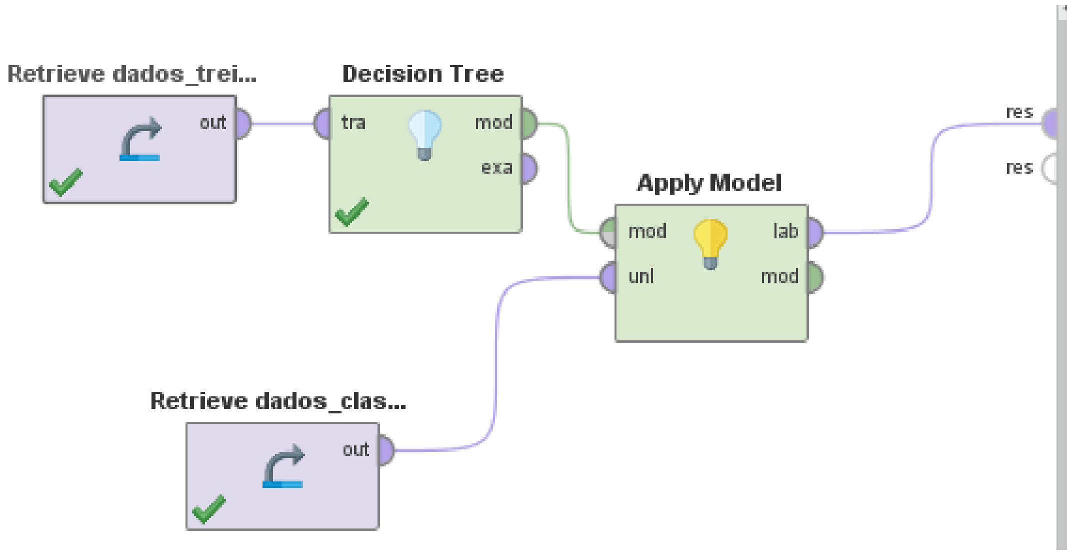


Figure 20 - Workflow of the Decision Tree model, gotten from RapidMiner results

5.5.2 Support Vector Machine Model

Support Vector Machine (SVM) is a non-probabilistic binary linear classifier that, given a set of inputted tested data (with two possible states), uses it as a sample to classify unknown data (see section 3.6.2). Figure 21 represents of the SVM model. It should be noted that, as in the Decision Tree model, both databases used were previously saved with the output from the data preparation step showed (section 5.3.1).

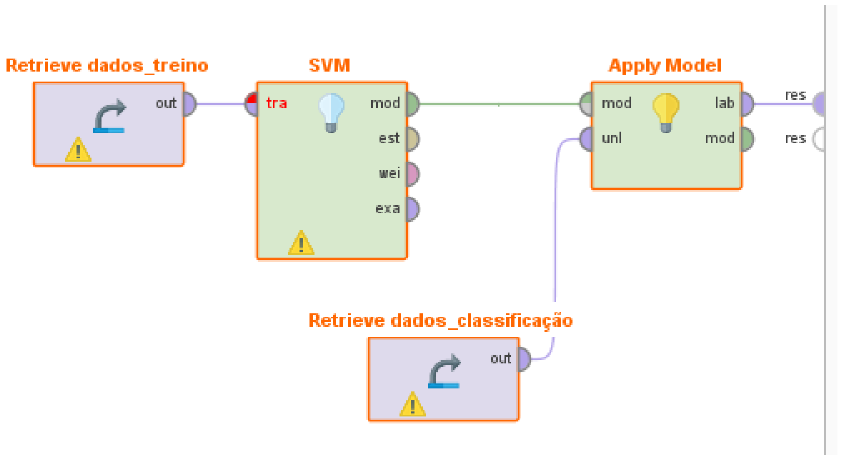


Figure 21 - Workflow of the SVM model, gotten from RapidMiner results

As seen in section 3.6.2, two important parameters of SVM are the coefficient (*C*) and epsilon (ε), respectively responsible for the trade-off between complexity and proportion of non-separable samples (thus the lower the C value the more generic the model gets with the cost of misclassification some points), and responsible for defining the level of influence a single training example reaches (where the lower its value the farther its reach).

During the experiment, different parameter setups were used, later seen in section 5.6.4. However, unlike the Decision Tree model[8], to better predict the *C* and ε values a practical guide to SVM classification developed by Hsu et al. (2010) was followed [98]. Its implementation is shown in Figure 22, where the operator “Optimize Parameters (Grid)” is used to execute the cross-validation test using all the combinations of values for the selected parameters (*C* and ε) and obtain the best performance values.

[8] This was not done to the DT parameters because no guide was found.

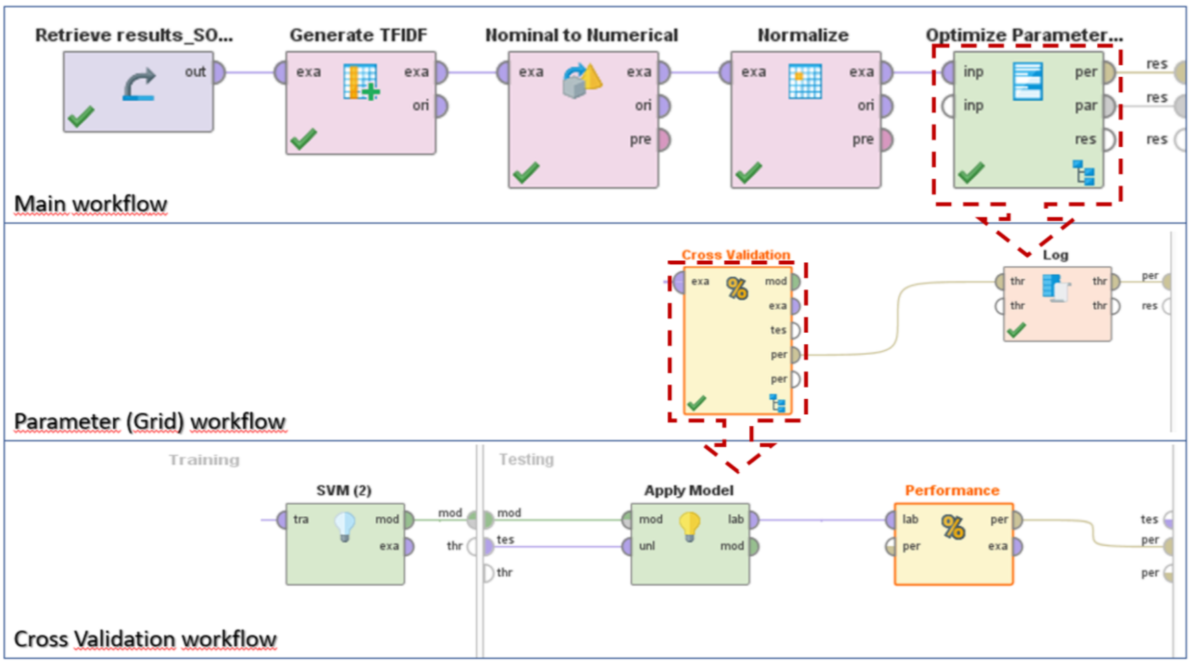


Figure 22 - SVM parameters calculation workflow, gotten from RapidMiner results

5.6 Classification Models Experiments

To ascertain that the models created fit the intended data analysis, tests were performed on the used models, namely a performance test and a validation test.

It is important to note that each model has various parameters, which influence the test results. Thereby, various tests were performed, using different parameters, to find the best parametrization for the models.

In both the performance test and the validation test, the data used was the sample with known behavior ("APP_ID"), which accounted for 100% of the collected data. Because the data type is (bi)nominal, the pruning, a machine learning technique that removes unimportant information, was not made use of in the tests.

5.6.1 Performance Test

Primarily, to test the accuracy of the model prediction a performance operator was used, in this step the test model was built and trained using only the known data, i.e. it tested if the flow came from a malicious application or from a normal application. One should be aware that this test information only validates the ability of the model to reproduce answers to known questions, which represents the training error. Figure 23 shows an example of the workflow used for the performance test of the Decision Tree model.

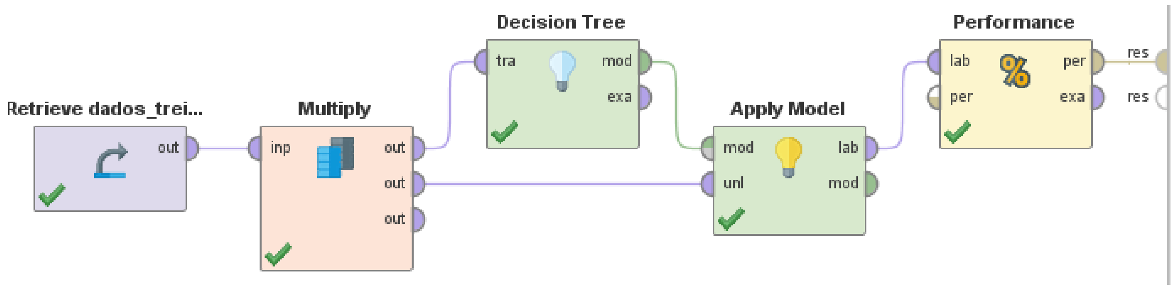


Figure 23 - Workflow of the evaluation performance of the Decision Tree model, gotten from RapidMiner results

The example of the SVM workflow performance test model is seen in Figure 24.

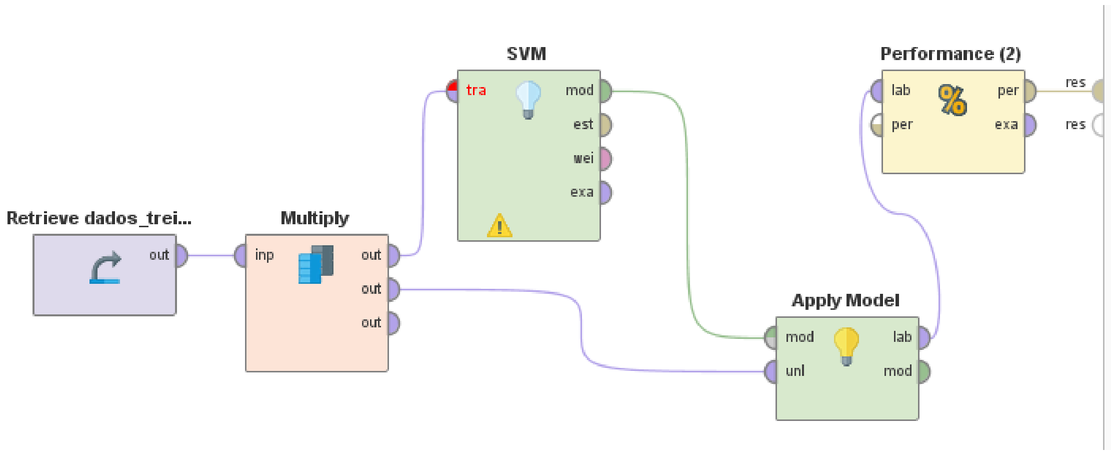


Figure 24 - Workflow of the evaluation performance on the SVM model, gotten from RapidMiner results

5.6.2 Validation Test

After testing the model performance, a validation test was performed to see how the model would behave when faced with unknown questions (i.e. unknown data set), which yields the prediction error of the model. To ensure that this test results are statistical significant estimators of the models' performance, and not "random luck" on the collected data, the 10-fold cross validation method was resorted to.

The cross-validation test takes the given inputted data and divides it into random samples of data subsets. The number of subsets of data is defined by the number of test folds (x-folds) defined, in this case 10 folds were used. It builds a model[9] on 9 of the data subsets and keeps 1 of the data subsets for testing and measures the performance of the model. To test the robustness of the model and avoid overfitting of the data subset used, the cross-validation test iterates x-fold times the testing data subset (i.e. changes the defined testing subset) and repeatedly calculates the model performance. Finally, it calculates the model accuracy by the performance average of each of the iterations done [99].

This step was performed on the Decision Tree and SVM classification models. Figure 25 shows an example of the workflow used for the SVM model validation. It should be noted that the parameters' definitions where the same in both models tests.

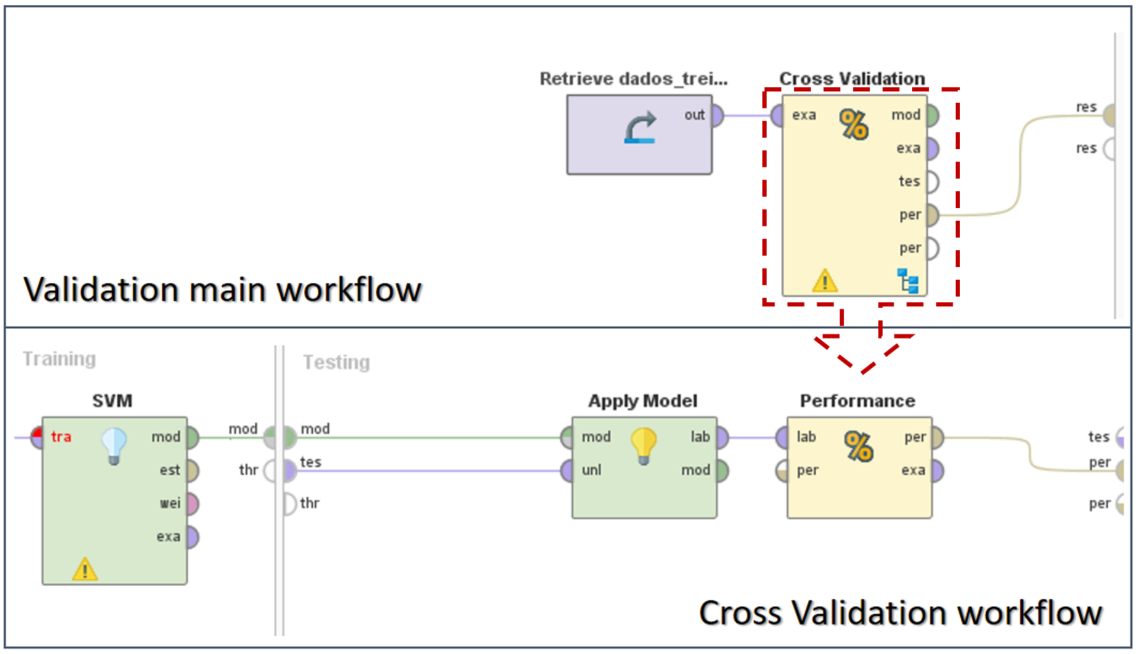


Figure 25 - Workflow of the validation performance SVM model, gotten from RapidMiner results

5.6.3 DT Model Experiment

When classifying the data, Decision Tree models use different attributes (features), found in the inputted data, to continually split the data until it reaches a prediction on the data label.

The model splits the data attributes based on the specified "Criterion" parameter (Table 3 shows the different captions for the "Criterion" parameter) and the maximum number of splits carried out is outlined by the "Maximal Depth" value.

[9] The model is built based on the model in the test.

Table 3 - Caption of the criterion parameters

Criterion	**Caption** [95]
Gini_index	"This is a measure of impurity of an *ExampleSet* (inputted data). Splitting on a chosen attribute gives a reduction in the average gini index of the resulting subsets."
Accuracy	"Such an attribute is selected for split that maximizes the accuracy of the whole Tree."
Information_gain	"The entropy of all the attributes is calculated. The attribute with minimum entropy is selected for split. This method has a bias towards selecting attributes with a large number of values."
Gain_ratio	"It is a variant of information gain. It adjusts the information gain for each attribute to allow the breadth and uniformity of the attribute values."

To test the model behavior with different parameter definitions, the performance tests were done using different "Criterion" parameters and different "Maximal Depth" values, Table 4 are specified the parameters used.

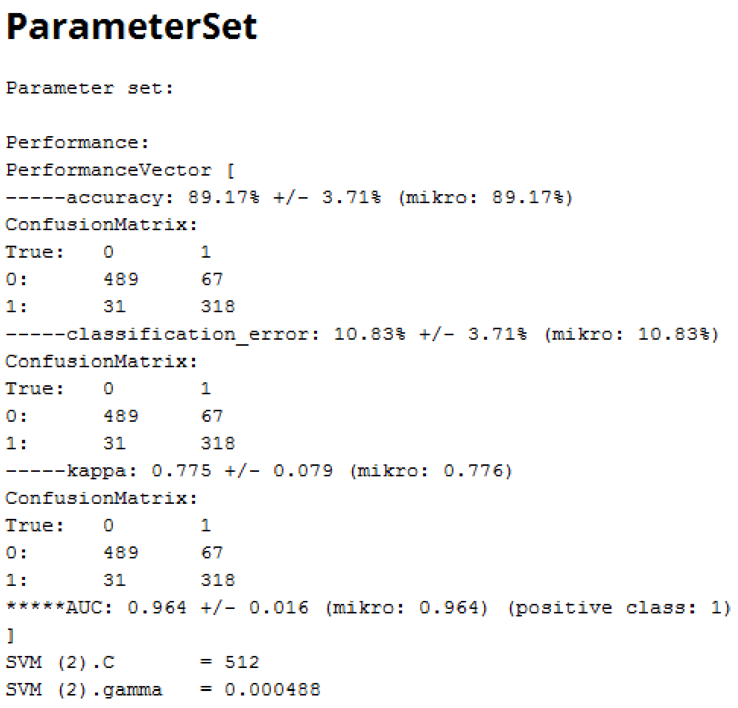
Unlike for the SVM model, for the DT model no parameterization model was found, therefore initially all the available "Criterion" parameters were used during the test and the "Maximal Depth" values were randomly chosen. After some "trial and error" tests performed, the parameters were adjusted per improvements seen in the results obtained.

Table 4 - Decision Tree algorithm parametrization' details

Parameters	**Test 1**	**Test 2**	**Test 3**	**Test 4**	**Test 5**	**Test 6**
Criterion	Gini_index	Gini_index	Gini_index	Accuracy	Gain_ratio	Information_gain
Maximal Depth	2	10	30	40	40	40
	Test 7	**Test 8**	**Test 9**	**Test 10**	**Test 11**	**Test12**
Criterion	Gini_index	Gini_index	Accuracy	Gini_index	Gini_index	Gini_index
Maximal Depth	40	70	70	75	85	100

5.6.4 SVM Model Experiment

As stated before (in section 5.5.2), to better parameterize the SVM model a value prediction model was applied. The outputted calculation results values can be seen in Figure 26, with a AUC (Area Under the Curve) performance of 0.964 +/- 0.016. The AUC refers to the area under the ROC (Receiver Operator Characteristic) which is a graph used as the baseline to assess whether the model is useful [100]. It is important to remember that this performance value is only for the model used to calculate the first C and ε values and it is independent from the evaluation of the SVM model.


ParameterSet

```
Parameter set:

Performance:
PerformanceVector [
-----accuracy: 89.17% +/- 3.71% (mikro: 89.17%)
ConfusionMatrix:
True:   0       1
0:      489     67
1:      31      318
-----classification_error: 10.83% +/- 3.71% (mikro: 10.83%)
ConfusionMatrix:
True:   0       1
0:      489     67
1:      31      318
-----kappa: 0.775 +/- 0.079 (mikro: 0.776)
ConfusionMatrix:
True:   0       1
0:      489     67
1:      31      318
*****AUC: 0.964 +/- 0.016 (mikro: 0.964) (positive class: 1)
]
SVM (2).C       = 512
SVM (2).gamma   = 0.000488
```

Figure 26 - Parameters calculation results outputted values, gotten from RapidMiner results

The parameters' calculation model was used as a reference to start the SVM model testing (Test 1,Table 5), however to find comparison of the C and epsilon (ε) impact, the tests that followed were performed using both the *RapidMiner* default values of the testing software and the calculated parameters, changing the kernel used. Account should be taken that, by variating the kernel used, i.e. by altering the mathematical expression used to divide the data, the model behavior and thereby the output results will also vary. All the used parameters values are specified in Table 5.

The *neural* kernel and *Gaussian Combination* kernel tests were initial considered to also be run, but the used machine resources limitations rendered them impossible, so they were not run.

Table 5 - SVM algorithm parametrization' details

Parameters	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6
C	512	0.0	512	0.0	512	0.0
ε	0.000488	0.001	0.000488	0.001	0.000488	0.001
Kernel	Linear	Linear	Radial	Radial	Polynomial	Polynomial
	Test 7	**Test 8**	**Test 9**	**Test 10**	**Test 11**	**Test12**
C	512	0.0	512	0.0	512	0.0
ε	0.000488	0.001	0.000488	0.001	0.000488	0.001
Kernel	Anova	Anova	Epachnenikok	Epachnenikok	Multiquadric	Multiquadric

5.6.5 Evaluation

Data mining models need to have the greatest possible acuity. Hence, for the models to be accepted as reliable for use in real cases, certain specifications must be met. In the scope of this problem, the time and memory constraints during the testing were not considered. The main criteria defined was the models' correct prediction ability, i.e. its reliability. Therefore, the evaluation parameters are Accuracy, Precision, and Recall. To ascertain these values from the cross-validation test, the data from the confusion matrix was also collected, as is shown in Table 6, to compare the actual classes of samples (correctly and incorrectly) defined against the predicted classes.

Table 6 - Confusion matrix

		Actual Class		**Class Precision**
		Normal (N)	**Malicious (M)**	
Predicted class	**Normal**	True Normal (TN)	False Malicious (FM)	$Precision_{Normal}$
	Malicious	False Normal (FN)	True Malicious (TM)	$Precision_{Malicious}$
Class Recall		$Recall_{Normal}$	$Recall_{Malicious}$	

Each of the evaluation parameters represents a percentage value that estimates the model reliability, respectively: the accuracy, Eq. 5, defines the ability of the model to correctly predict the class label of unknown data; the precision, Eq. 6 and Eq. 7, defines the percentage of relevant results (quality); and, the recall, Eq. 8 and Eq. 9, defines the percentage of relevant results (quantity) [101].

$$Accuracy = \frac{TN + \mathrm{TM}}{N + M} \qquad (Eq. 5)$$

$$Precision_{Malicious} = \frac{\mathrm{TM}}{FN + TM} \qquad (Eq. 6)$$

$$Precision_{Normal} = \frac{\mathrm{TN}}{TN + FM} \qquad (Eq. 7)$$

$$Recall_{Malicious} = \frac{\mathrm{TM}}{FM + TM} \qquad (Eq. 8)$$

$$Recall_{Normal} = \frac{TN}{TN + FN} \qquad (Eq. 9)$$

In addition to the confusion matrix, during the cross-validation test *RapidMiner* presents a global average value for the Accuracy, the Recall and the Precision values, i.e. the average of each parameter for the malicious and normal results.

Table 7 shows the percentage values defined for evaluation of the performance and validation of the models, note that these values represent the global average. The minimum accepted percentage value of the models was decided based on the performance values found in the literature review (in Chapter 4) and other security performance studies done using cross-validation [102] [103].

Table 7 - Defined percentage values for the models evaluation

	Testing Model[10]	**Validating Model**
Accuracy	Higher that 90%	higher than 80%
Precision	Higher that 90%	higher than 80%
Recall	Higher that 90%	higher than 70%

An additional information calculated by *RapidMiner* is the standard deviation (σ) value, for both the Accuracy and Recall values, which represents the amount of variation or dispersion of a set of data values (i.e. the lower this value the more stable the model is) [101].

[10] For the Decision Tree performance test the model only evaluates the "Accuracy".

6. RESULTS

As previously stated (in section 5.6.5), *RapidMiner* gives a global average value for each evaluated parameter, thereby those results were first gathered. The global results obtained from the performance and validation tests in the Decision Tree (DT) are shown in Table 8. To ease the observational analysis of the tables, the best results obtained are highlighted (in yellow).

Table 8 - Decision Tree model tests results

Model	Test	Test type	Accuracy (%)(+/-σ)	Precision (%)	Recall (%)(+/-σ)
DT	**Test 1**	Performance	72.87	-	-
		Validation	72.87 (+/- 3.55)	100.00 (+/- 0.00)	32.90 (+/- 8.55)
	Test 2	Performance	81.52	-	-
		Validation	78.06 (+/- 3.78)	100.00 (+/- 0.00)	45.70 (+/- 9.42)
	Test 3	Performance	92.29	-	-
		Validation	86.57% +/- 3.07	98.47% +/- 2.35	67.70% +/- 6.83
	Test 4	Performance	91.09	-	-
		Validation	84. 57 (+/- 3.70)	93.65 (+/- 6.12)	66.70 (+/- 8.90)
	Test 5	Performance	90.96	-	-
		Validation	85.24 (+/- 3.88)	98.29 (+/- 2.62)	64.42 (+/- 8.73)
	Test 6	Performance	95.35	-	-
		Validation	87.36 (+/- 2.75)	98.55 (+/- 2.22)	69.67 (+/- 6.15)
	Test 7	Performance	95.08	-	-
		Validation	87.36 (+/- 2.75)	98.55 (+/- 2.22)	69.67 (+/- 6.15)
	Test 8	Performance	99.20	-	-
		Validation	89.36 (+/- 3.38)	98.25 (+/- 3.38)	89.36 (+/- 3.38)
	Test 9	Performance	95.48	-	-
		Validation	86.69 (+/- 4.01)	93.87 (+/- 6.08)	72.25 (+/- 10.53)
	Test 10	Performance	99.20	-	-
		Validation	89.36 (+/- 3.38)	98.25 (+/- 2.76)	74.90 (+/- 7.68)
	Test 11	Performance	99.20	-	-
		Validation	89.36 (+/- 3.38)	98.25 (+/- 2.76)	74.90 (+/- 7.68)
	Test 12	Performance	99.20	-	-
		Validation	89.36 (+/- 3.38)	98.25 (+/- 2.76)	74.90 (+/- 7.68)

The global results obtained from the performance and validation tests in the SVM model are shown in Table 9. Similarly to the table of the DT model results, the best results are also highlighted (in yellow).

Table 9 - SVM model tests results

Model	Test	Test type	Accuracy (%) (+/-σ)	Precision (%)	Recall (%) (+/-σ)
SVM	**Test 1**	Performance	99.20	100.0	98.03
		Validation	65.31 (+/- 11.73)	95.74	15.00 (+/- 30.01)
	Test 2	Performance	98.80	99.66	97.37
		Validation	65.31 (+/- 11.73)	95.74	15.00 (+/- 30.01)
	Test 3	Performance	99.20	100.00	98.03
		Validation	61.84 (+/- 4.87)	100.00	5.67 (+/- 11.55)
	Test 4	Performance	99.20	100.00	98.03
		Validation	61.84 (+/- 4.87)	100.00	5.67 (+/- 11.55)
	Test 5	Performance	70.61	58.03	98.68
		Validation	60.77 (+/- 2.70)	58.18	10.67 (+/- 21.33)
	Test 6	Performance	88.96	100.00	72.70
		Validation	60.77 (+/- 2.83)	65.52	6.33 (+/- 12.69)
	Test 7	Performance	98.80	97.73	99.34
		Validation	65.04 (+/- 11.17)	85.96	16.33 (+/- 32.68)
	Test 8	Performance	59.57	unknown	0.00
		Validation	-	-	-
	Test 9	Performance	99.20	100.00	98.03
		Validation	61.84 (+/- 4.87)	100.00	5.67 (+/- 11.55)
	Test 10	Performance	99.20	100.00	98.03
		Validation	61.84 (+/- 4.87)	100.00	5.67 (+/- 11.55)
	Test 11	Performance	59.57	unknown	0.00
		Validation	**-**	**-**	**-**
	Test 12	Performance	59.57	unknown	0.00
		Validation	-	-	-

The best result tests, obtained for each model, are discriminated in the confusion matrix (Table 10), where the DT and SVM models are respectively shown. Highlighted in yellow are the best true predicted scores and in blue the lowest true predictions.

Table 10 - Confusion matrix of the models best validation test results

Model	**Test**	**% predicted as "normal" (TN)**	**% successful "normal" predictions**	**% predicted as "malicious" (TM)**	**% successful "malicious" predictions**
DT	Test 8	94.62	83.67	75.06	91.17
	Test 9	92.12	88.01	78.18	88.01
	Test 10	94.62	84.10	75.85	91.25
	Test 11	94.62	84.97	77.40	91.41
	Test 12	94.62	85.12	77.66	91.44
SVM	Test 1	99.55	63.26	14.80	95.74
	Test 2	99.55	63.26	14.80	95.74
	Test 7	98.21	63.31	16.12	85.96

% successful "normal/malicious" predictions	Percentage of predictions, for each class, the models managed to made.
% predicted as "normal/malicious" (TN/TM)	Percentage of correct predictions, for each class, the models managed to made.

6.1 Discussion

The discussion performed was divided into three parts. The first part focuses on the results of the DT model, which is subdivided into a direct observation analysis and an examination of the model tests. Then, the SVM parameters optimization testing and model results is discussed. Finally, a comparison of the performance between the models is done.

6.1.1 Decision Tree Model Discussion

Direct Observation

For a preliminary examination of the DT model results an example was chosen from the performed tests, and its output is analyzed. The graphical representation used is shown in Figure 27, this figure corresponds to an output gotten during the experimental tests.

From the branches shown in the figure), it can be interpreted that a flow will only be normal if its normalized value is equal to or lower than 7.864[11], meaning that above that value are the malicious flows.

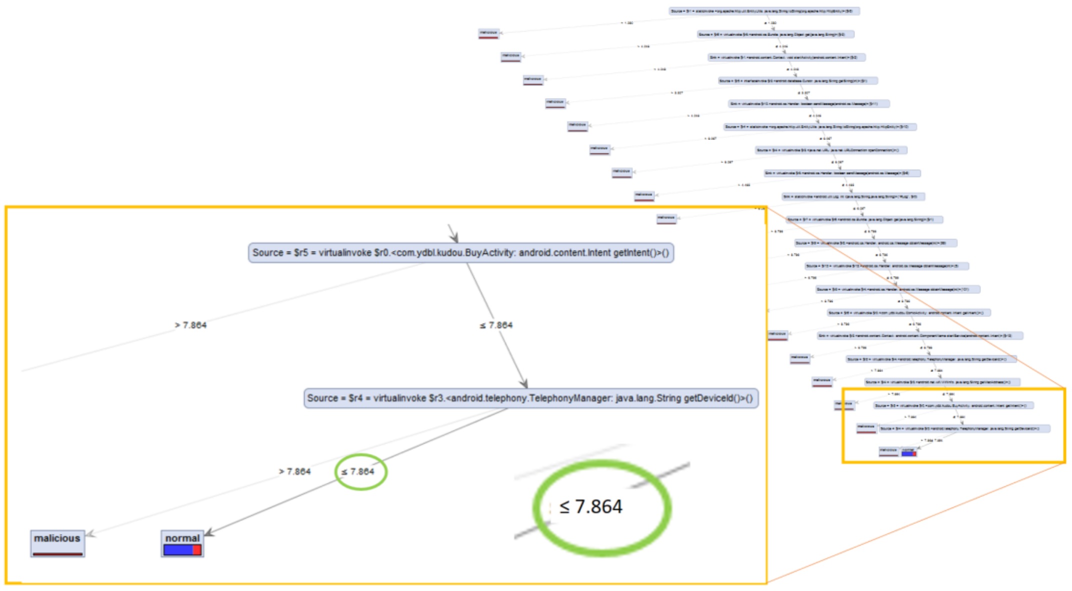
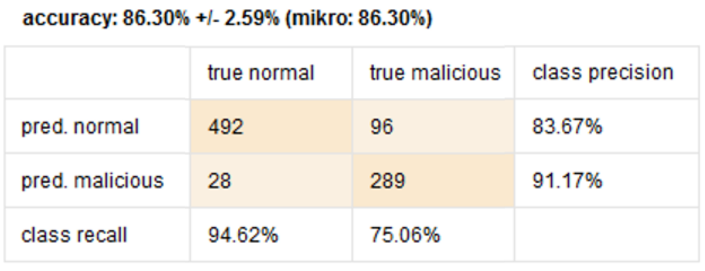

Source = $r5 = virtualinvoke $r0.<com.ydbl.kudou.BuyActivity: android.content.Intent getIntent()>()
> 7.864
≤ 7.864
Source = $r4 = virtualinvoke $r3.<android.telephony.TelephonyManager: java.lang.String getDeviceId()>()
> 7.864
≤ 7.864
≤ 7.864
malicious
normal

Figure 27 - Output from the training Tree Decision model, gotten from RapidMiner results

An example of the outputted values gotten from *RapidMiner* during the validation tests is presented in Figure 28. These values represent the output from the Decision Tree model validation test (Test 8). It is noteworthy that "pred. normal" stands for predicted normal and "pred. malicious" stands for predicted malicious. Because both images were gotten at different times during the project, it should be noted that figures 27 and 28 represent two different test results.

In this example, focusing on the malicious flows predictions, the model managed to predict 75.06% true malicious flows. However, in 9 cases of malicious flow predictions one of the flows was a normal flow ("pred. malicious" of 91.17%).



accuracy: 86.30% +/- 2.59% (mikro: 86.30%)

	true normal	true malicious	class precision
pred. normal	492	96	83.67%
pred. malicious	28	289	91.17%
class recall	94.62%	75.06%	

Figure 28 - Illustration of the outputted values of the DT model validation test (Test 8),

[11] Numeric value calculated by the Decision Tree algorithm, based on the criterion parameter definition.

gotten from RapidMiner results

Decision Tree Model Test Results

Based on the analysis of the results obtained during the tests performed on the DT model, it can be verified that in terms of accuracy and precision almost all the tests performed (from Test 3 to Test 12) comply with the minimal values defined, and the Tests 8 to 12 are quite satisfactory since they all comply with the minimum specifications.

Individually, Tests 1 and Test 2 obtained the best precision (=100%), however this value could be due to an overfitting of the models for the parameters used in this tests. Whereas Tests 8 to 12 obtained the best accuracy (= 89.36 +/- 3.38) and Test 8 had the best recall value (=89.36% +/-3.38).

Test 8 parameters obtained the best DT model performance, using the criterion defined as “Gini_index” and a maximal Depth of 70, with accuracy of 89.36%, precision of 98.25%, recall of 89.36% and a standard deviation of 3.38. This model fits the data well, however because of the depth used it may not fit different data sets.

From the confusion matrix (Table 10), it is possible to differentiate that although Test 8 had the best results when analyzing the global values (on Table 8), it was the Test 9 that had the best score (78.18%) when predicting malicious flows, whereas Test 8 afforded the lowest true malicious prediction score (75.06%). Test 9 had the lowest true normal prediction percentage (92.12%) and all the others scored 94.62% true normal predictions.

6.1.2 Support Vector Machine Model Discussion

The SVM model did not produce any graphical representation that could be analyzed and its confusion matrix is similar to the DT confusion matrix. Thereby, for the SVM results only the results on the tests were considered.

Unfortunately, the SVM model results were not satisfactory, unlike what the performance results indicated in the cross-validation tests. None of the performed tests complied with the minimum specifications. Among all of them, Test 7 obtained the best results, with the parameters definition: C = 512, ε = 0.000488 and “Anova” kernel, and an accuracy of 65.04%, precision of 85.96%, recall of 16.33 %, standard deviation for accuracy of 11.17 and a standard deviation for recall of 32.68.

From the data in the confusion matrix it can be concluded that the model completely fails on detecting malicious behaviors, with the best true malicious score being of only 16.12% (Test 7).

6.1.3 Results Comparison

From the performance test results obtained, it seemed reasonable to assume that both models would have good results when analyzing new data, with both the DT and SVM model having, for most tests, more that 90% in accuracy results.

However, from the 10-fold cross validation test it was found that the SVM model never met the minimum values, with best test results presenting an accuracy of 65.04%, a precision of 85.96%, a recall of 16.33 % and standard deviations for accuracy and recall, respectively, of 11.17 and 32.68 which also mean the model is unstable.

The DT classification model seems to have a good response rate when detecting the data type used. From the 10-fold cross validation test results, it can be concluded that the model was a good fit for the data analyzed, with 33% of the tests meeting the minimum evaluation values defined. This means the model successfully found behavioral patterns in the data (both normal and malicious). However, the model was not generalized given that most maximal depth values used were over 40 (branches).

Between the best results obtained for each model and their standard deviation values, it can be concluded that the DT model (σ = 3.38) was more stable than the SVM model (σ = 32.68).

From the experiment, the DT model was a good classification model for the data obtained. It was shown that it is possible to distinguish, with some precision, malicious patterns on the data used.

7. Conclusion

Digital information security has been a great concern for government and enterprises, initially directed to the information in computers and servers. The widespread usability of mobile phone users to handle information made them into a target for malicious users, particularly the Android OS system.

Malicious software is constantly being developed and the intrusion techniques are increasingly sophisticated. Criminals, use different types of intrusion methods (i.e. Trojan horses, social engineering, etc.) trying to gain access to sensitive information, causing information leakage. To counteract these intrusions systems, security protection systems are constantly in the need of improvement and updating.

The research question goal was to assess whether it was possible to define measurable behavior patterns in malicious applications.

By collecting the outputted data from the *FlowDroid* application, flows from sources to sinks, and applying it into the DT and SVM models built, an analysis was made on the models' ability to identify behavioral patterns in the data and correctly classify it as either normal or malicious. To test the models both performance and validation tests were used.

The results obtained, from the DT model, indicate that it is possible to determine malicious flows in malicious applications and that the flows found share a level of similar attributes, which enabled the classification of behavior patterns between the applications.

The characterization and detection of tainted flows is an important feature on security applications. Particularly for companies and governments, were the loss of sensitive information can represent severe losses. Private users should also be more concerned with their own privacy leakages. However, more than having good security applications, there is the need to raise the users' awareness on the risks they expose themselves when installing doubtful applications and the negative impact that could mean for their privacy or on their device.

7.1 Project Limitations and Future Work

This dissertation has developed two classification models, a DT and SVM model, with the purpose of showing if these models could use the behavioral patterns found in the collected data and correctly classify the flows (data collected) into malicious or normal flows, however some limitations were found in this study.

A constraint of this study was the SourcesAndSinks.txt document used in the data collection phase. Because the document used translates the sources and sinks collected by the *FlowDroid* application's authors, it may be possible that from the analyzed applications for this study some flows in those applications were not found and thereby not considered in the data classification phase. This means that some information may be missing from the analyzed data set. For an improvement of this project, and greater accuracy of the results, an assessment on the sources and sinks per application needs to be done.

Based on the created DT model and with the obtained results, it is possible to make a prior identification of the malicious applications. Therefore, a future proposal for the continuation of the project is to expand the data collection and try other data mining techniques (such as data clustering algorithms, deep learning, or neural networks) to uncover exactly which patterns the malicious flows form. For example, to create a detection application that directly looks for this patterns when analyzing the system, making the detection faster and less resource consuming.

REFERENCES

[1] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, and S. Dolev, “Google Android: A State-of-the-Art Review of Security Mechanisms,” *arXiv Prepr. arXiv0912.5101*, 2009.

[2] Y. Zhou and X. Jiang, “Dissecting Android Malware: Characterization and Evolution,” in *2012 IEEE Symposium on Security and Privacy*, 2012, pp. 95–109.

[3] S. G. Cheetancheri, J. M. Agosta, D. H. Dash, K. N. Levitt, J. Rowe, and E. M. Schooler, “A distributed host-based worm detection system,” *Proc. 2006 SIGCOMM Work. Largescale attack Def. LSAD 06*, pp. 107–113, 2006.

[4] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, “Google android: A comprehensive security assessment,” *IEEE Secur. Priv.*, vol. 8, no. 2, pp. 35–44, 2010.

[5] W. H. Park, D. H. Kim, M. S. Kim, and N. Park, “A Study on Trend and Detection Technology for Cyber Threats in Mobile Environment,” in *2013 International Conference on IT Convergence and Security (ICITCS)*, 2013, pp. 1–4.

[6] M. Spreitzenbarth, F. C. Freiling, F. Echtler, T. Schreck, and J. Hoffmann, “Mobile-sandbox: Having a Deeper Look into Android Applications,” *Proc. 28th Annu. ACM Symp. Appl. Comput.*, pp. 1808–1815, 2013.

[7] K. Peffers, T. Tuunanen, M. Rothenberger, and S. Chatterjee, “A Design Science Research Methodology for Information Systems Research,” *J. Manag. Inf. Syst.*, vol. 24, no. January, pp. 45–77, 2008.

[8] P. K. Dixit, *Android*. Vikas Publishing House, 2014.

[9] S. Brähler, “Analysis of the Android Architecture,” *Os.Ibds.Kit.Edu*, p. 52, 2010.

[10] A. Shrivastava and P. Mahajan, “Android Security Enhancements,” *Android Tamer*, p. 7, 2015.

[11] S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A. Sadeghi, “XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks,” *Tech. Univ. Darmstadt, Tech. Rep.*, pp. 1–18, 2011.

[12] M. Heinl and I. Technology, “Android Security,” 2015.

[13] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, “A Study of Android Application Security.,” *USENIX Secur.*, vol. 39, no. August, pp. 21–21, 2011.

[14] J. J. F. DiMarzio, *Android - A Programmer’s Guide*. 2008.

[15] J. Kim, Y. Yoon, K. Yi, and J. Shin, “Scandal: Static Analyzer for Detecting Privacy Leaks in Android Applications,” *IEEE Work. Mob. Secur. Technol.*, pp. 1–10, 2012.

[16] elinux.org, “Android Binder,” *eLinux.org*, 2014.

[17] B. Kim *et al.*, “Evaluation of Android Dalvik Virtual Machine,” pp. 115–124, 2012.

[18] A. K. Parida, “Android Application Development for GPS Based Location Tracker.”

[19] W. Shin, S. Kiyomoto, K. Fukushima, and T. Tanaka, “A formal model to analyze the permission authorization and enforcement in the Android framework,” *Proc. - Soc. 2010 2nd IEEE Int. Conf. Soc. Comput. PASSAT 2010 2nd IEEE Int. Conf. Privacy, Secur. Risk Trust*, pp. 944–951, 2010.

[20] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, “Permission Evolution in the Android Ecosystem,” *ACSAC ’12 Proc. 28th Annu. Comput. Secur. Appl. Conf.*, no. April 2009, pp. 31–40, 2012.

[21] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, “Android permissions demystified,” *Proc. 18th ACM Conf. Comput. Commun. Secur. - CCS ’11*, p. 627, 2011.

[22] E. Chin, A. Felt, K. Greenwood, and D. Wagner, “Analyzing inter-application communication in Android,” *Proc. 9th ...*, pp. 239–252, 2011.

[23] N. Peiravian and X. Zhu, "Machine learning for Android malware detection using permission and API calls," *Proc. - Int. Conf. Tools with Artif. Intell. ICTAI*, pp. 300–305, 2013.

[24] C. Marforio, A. Francillon, and S. Capkun, "Application Collusion Attack on the Permission-Based Security Model and its Implications for Modern Smartphone Systems," pp. 1–16, 2011.

[25] Google Inc, "Android Developers." [Online]. Available: https://developer.android.com/about/dashboards/index.html.

[26] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon, "Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android," pp. 274–277, 2012.

[27] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android Permissions : User Attention , Comprehension , and Behavior," 2012.

[28] W. Shin, S. Kwak, S. Kiyomoto, K. Fukushima, and T. Tanaka, "A small but non-negligible flaw in the android permission scheme," *Proc. - 2010 IEEE Int. Symp. Policies Distrib. Syst. Networks, Policy 2010*, pp. 107–110, 2010.

[29] P. Faruki *et al.*, "Android Security: A Survey of Issues, Malware Penetration, and Defenses," *IEEE Commun. Surv. Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015.

[30] S. Bugiel, L. Davi, A. Dmitrienko, S. Heuser, A.-R. Sadeghi, and B. Shastry, "Practical and lightweight domain isolation on Android," *ACM CCS Work. Secur. Priv. Smartphones Mob. Devices*, p. 51, 2011.

[31] D. Barbará, J. Couto, S. Jajodia, L. Popyack, and N. Wu, "ADAM: Detecting Intrusions by Data Mining," *Proc. IEEE Work. Inf. Assur. Secur.*, no. June, pp. 11–16, 2001.

[32] M. Nauman, S. Khan, and X. Zhang, "Apex: extending Android permission model and enforcement with user-defined runtime constraints," *Asiaccs*, pp. 328–332, 2010.

[33] D. Octeau *et al.*, "Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis," *USENIX Secur. Symp.*, pp. 543–558, 2013.

[34] L. Khan, M. Awad, and B. Thuraisingham, "A new intrusion detection system using support vector machines and hierarchical clustering," *VLDB J.*, vol. 16, no. 4, pp. 507–521, 2007.

[35] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," *Proc. 1st ACM Work. Secur. Priv. smartphones Mob. devices - SPSM '11*, pp. 3–14, 2011.

[36] V. N. Cooper, H. Shahriar, and H. M. Haddad, "A survey of android malware characterisitics and mitigation techniques," *ITNG 2014 - Proc. 11th Int. Conf. Inf. Technol. New Gener.*, pp. 327–332, 2014.

[37] M. Chandramohan and H. B. K. Tan, "Detection of mobile malware in the wild," *Computer (Long. Beach. Calif).*, vol. 45, no. 9, pp. 65–71, 2012.

[38] Z. Trabelsi, K. Hayawi, A. Al Braiki, and S. S. Mathew, *Network Attacks and Defenses: A Hands-on Approach*, Ilustrada. CRC Press, 2012.

[39] R. Bejtlich, "Extrusion Detection: Security Monitoring for Internal Intrusions," p. 416, 2005.

[40] A. Karim, R. Salleh, M. Shiraz, S. Shah, I. Awan, and N. Anuar, "Botnet detection techniques: review, future trends, and issues," *J. Zhejiang Univ. C (Computers Electron.*, vol. 15, no. 11, pp. 943–983, 2014.

[41] R. Braga, E. Mota, and A. Passito, "Lightweight DDoS flooding attack detection using NOX/OpenFlow," *Proc. - Conf. Local Comput. Networks, LCN*, pp. 408–415, 2010.

[42] M. La Polla, F. Martinelli, and D. Sgandurra, "A Survey on Security for Mobile Devices," *IEEE Commun. Surv. Tutorials*, vol. 15, no. 1, pp. 446–471, 2013.

[43] J. Berdajs and Z. Bosnic, "Extending applications using an advanced approach to DLL injection and API hooking," *Softw. - Pract. Exp.*, vol. 39, no. 7, pp. 701–736, 2010.

[44] I. Project, "Android RAM analysis."

[45] L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Over-the-Air Cross-platform Infection for Breaking mTAN-based Online Banking Authentication," *Black Hat Abu Dhabi*, pp. 1–12, 2012.

[46] V. Thing and P. Subramaniam, "Mobile Phone Anomalous Behaviour Detection forReal-time Information Theft Tracking," *CYBERLAWS 2011, …*, no. c, pp. 7–11, 2011.

[47] A. Patel, M. Taghavi, K. Bakhtiyari, and J. Celestino Júnior, "An intrusion detection and prevention system in cloud computing: A systematic review," *J. Netw. Comput. Appl.*, vol. 36, no. 1, pp. 25–41, 2013.

[48] M. A. Faysel and S. S. Haque, "Towards cyber defense: research in intrusion detection and intrusion prevention systems," *IJCSNS Int. J. Comput. Sci. Netw. Secur.*, vol. 10, no. 7, pp. 316–325, 2010.

[49] I. Butun, S. D. Morgera, and R. Sankar, "A Survey of Intrusion Detection Systems in Wireless Sensor Networks," *IEEE Sens. J.*, vol. 14, no. 5, pp. 1370–1379, 2014.

[50] S. Yi, C. Li, and Q. Li, "A Survey of Fog Computing: Concepts, Applications and Issues," *Proc. 2015 Work. Mob. Big Data - Mobidata '15*, pp. 37–42, 2015.

[51] H.-J. Liao, C.-H. Richard Lin, Y.-C. Lin, and K.-Y. Tung, "Intrusion detection system: A comprehensive review," *J. Netw. Comput. Appl.*, vol. 36, no. 1, pp. 16–24, Jan. 2013.

[52] E. Tombini, H. Debar, L. Mé, and M. Ducassé, "A serial combination of anomaly and misuse IDSes applied to HTTP traffic," *Proc. - Annu. Comput. Secur. Appl. Conf. ACSAC*, pp. 428–437, 2004.

[53] J. Zhang and M. Zulkernine, "A hybrid network intrusion detection technique using random forests," in *First International Conference on Availability, Reliability and Security (ARES'06)*, 2006, p. 8 pp.-pp.269.

[54] D. Barbara and C. Kamath, "Proceedings of the Third SIAM International Conference on Data Mining," Philadelphia SIAM, Society for Industrial and Applied Mathematics, 2003, p. 347.

[55] R. Berthier, W. H. Sanders, and H. Khurana, "Intrusion Detection for Advanced Metering Infrastructures: Requirements and Architectural Directions," *2010 First IEEE Int. Conf. Smart Grid Commun.*, pp. 350–355, 2010.

[56] B. Amos, H. Turner, and J. White, "Applying machine learning classifiers to dynamic android malware detection at scale," *2013 9th Int. Wirel. Commun. Mob. Comput. Conf. IWCMC 2013*, pp. 1666–1671, 2013.

[57] S. K. Dash *et al.*, "DroidScribe : Classifying Android Malware Based on Runtime Behavior," *Mob. Secur. Technol.*, 2016.

[58] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: Behavior-Based Malware Detection System for Android," *Proc. 1st ACM Work. Secur. Priv. smartphones Mob. devices - SPSM '11*, p. 15, 2011.

[59] L. Li *et al.*, "I know what leaked in your pocket: uncovering privacy leaks on Android Apps with Static Taint Analysis," *CoRR*, no. April 2014, 2014.

[60] S. Arzt *et al.*, "FlowDroid : Precise Context , Flow , Field , Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," *PLDI '14 Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implement.*, pp. 259–269, 2014.

[61] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy : Semantics-Based Detection of Android Malware Through Static Analysis," *Fse*, pp. 16–22, 2014.

[62] S. Hwang, S. Cho, and S. Park, "Keystroke dynamics-based authentication for mobile devices," *Comput. Secur.*, vol. 28, no. 1–2, pp. 85–93, 2009.

[63] D. J. Wu, C. H. Mao, T. E. Wei, H. M. Lee, and K. P. Wu, "DroidMat: Android malware detection

through manifest and API calls tracing," *Proc. 2012 7th Asia Jt. Conf. Inf. Secur. AsiaJCIS 2012*, pp. 62–69, 2012.

[64] C. Estan, S. Savage, and G. Varghese, "Automatically inferring patterns of resource consumption in network traffic," in *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications - SIGCOMM '03*, 2003, pp. 137–148.

[65] L. Deshotels, V. Notani, and A. Lakhotia, "DroidLegacy: Automated Familial Classification of Android Malware," *Proc. ACM SIGPLAN Progr. Prot. Reverse Eng. Work. 2014*, pp. 1–12, 2014.

[66] P. S. D. J. Hand, Heikki Mannila, *Principles of Data Mining*. Cambridge, Mass : MIT Press, 2000.

[67] M. K. Jiawei Han, Jian Pei, *Data Mining: Concepts and Techniques*. .

[68] T. M. M. R.S. Michalski, J.G. Carbonell, *Machine Learning: An Artificial Intelligence Approach*. pringer Science & Business Media, 2013.

[69] J. Zhou and J. Tian, "Predicting corporate financial distress based on integration of decision tree classification and logistic regression." pp. 2–4, 2007.

[70] M. A. Friedl and C. E. Brodley, "Decision tree classification of land cover from remotely sensed data," *Remote Sensing of Environment*, vol. 61, no. 3. pp. 399–409, 1997.

[71] MIT OpenCourseWare, *16. Learning: Support Vector Machines*. .

[72] L. Liu, Z. Li, Ling Xu, and H. Chen, "A Security Event Management Framework Using Wavelet and Data-Mining Technique," *IEEE*, vol. 0, pp. 1257255–1257255, 2006.

[73] C. M. Ou and C. R. Ou, "Immunity-inspired host-based intrusion detection systems," *Proc. - 2011 5th Int. Conf. Genet. Evol. Comput. ICGEC 2011*, pp. 283–286, 2011.

[74] O. Koucham, "Host Intrusion Detection using System Call Argument-based Clustering combined with Bayesian Classification," pp. 1010–1016, 2015.

[75] B. K. Addagada, "Intrusion Detection in Mobile Phone Systems Using Data Mining Techniques," 2010.

[76] S. S. Khurana, D. Bansal, and S. Sofat, "Recovery based architecture to protect HIDS log files using time stamps," *J. Emerg. Technol. Web Intell.*, vol. 2, no. 2, pp. 110–114, 2010.

[77] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "'Andromaly': a behavioral malware detection framework for android devices," *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, Feb. 2012.

[78] M. Sun, M. Zheng, J. C. S. Lui, and X. Jiang, "Design and implementation of an Android host-based intrusion prevention system," *Proc. 30th Annu. Comput. Secur. Appl. Conf. - ACSAC '14*, pp. 226–235, 2014.

[79] L. Yan and H. Yin, "Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," *Proc. 21st USENIX Secur. Symp.*, p. 29, 2012.

[80] W. Enck, N. Carolina, and P. Gilbert, "TaintDroid : An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," vol. 32, no. 2, 2014.

[81] Anthony Desnos and P. Lantz, "DroidBox: An Android Application Sandbox for Dynamic Analysis." .

[82] C. Zheng, S. Zhu, S. Dai, G. Gu, and X. Gong, "SmartDroid : an Automatic System for Revealing UI-based," pp. 93–104, 2012.

[83] C. D. G. Alexander G. Gounares, "Control Flow Graph Driven Operating System," US 20120324454 A1, 2012.

[84] J. Kinable and O. Kostakis, "Malware classification based on call graph clustering," *J. Comput. Virol.*, vol. 7, no. 4, pp. 233–245, 2011.

[85] J. Huang, X. Zhang, L. Tan, P. Wang, and B. Liang, “AsDroid : Detecting Stealthy Behaviors in Android Applications by User Interface and Program Behavior Contradiction,” pp. 1036–1046, 2014.

[86] S. Rasthofer, S. Arzt, and E. Bodden, “A Machine-learning Approach for Classifying and Categorizing Android Sources and Sinks,” *Proc. 2014 Netw. Distrib. Syst. Secur. Symp.*, no. February, pp. 23–26, 2014.

[87] Z. Yang and M. Yang, “LeakMiner: Detect Information Leakage on Android with Static Taint Analysis,” *IEEE*, 2012.

[88] M. V. C. Fritz, “FlowDroid : A Precise and Scalable Data Flow Analysis for Android,” 2013.

[89] J. Jang, D. Brumley, and S. Venkataraman, “BitShred : Feature Hashing Malware for Scalable Triage and Semantic Analysis,” *Proc. 18th ACM Conf. Comput. Commun. Secur.*, pp. 309–320, 2011.

[90] R. Perdisci, D. Ariu, and G. Giacinto, “Scalable fine-grained behavioral clustering of HTTP-based malware,” *Comput. Networks*, vol. 57, no. 2, pp. 487–500, 2013.

[91] M. Thomas and A. Mohaisen, “Kindred Domains : Detecting and Clustering Botnet Domains Using DNS Traffic,” *WWW Work.*, pp. 707–712, 2014.

[92] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos, “Malicious Android applications in the enterprise: What do they do and how do we fix it?,” *Proc. - 2012 IEEE 28th Int. Conf. Data Eng. Work. ICDEW 2012*, pp. 251–254, 2012.

[93] P. Beaucamps *et al.*, “Behavior Abstraction in Malware Analysis - Extended Version To cite this version : Behavior Abstraction in Malware Analysis,” 2010.

[94] RapidMiner, “RapidMiner.” [Online]. Available: https://rapidminer.com/.

[95] RapidMiner, “RapidMiner Documentation.” [Online]. Available: http://docs.rapidminer.com/studio/operators/.

[96] Google Inc, “Google Play.” [Online]. Available: https://play.google.com/store/apps.

[97] Mila, “Contagium - malware dump.” [Online]. Available: http://contagiominidump.blogspot.pt/.

[98] C.-W. Hsu, C.-C. Chang, and C.-J. Lin, “A Practical Guide to Support Vector Classification,” *BJU Int.*, vol. 101, no. 1, pp. 1396–400, 2010.

[99] L. Liu and M. T. Özsu, *Encyclopedia of Database Systems*. 2009.

[100] C. Marzban, “The ROC Curve and the Area under It as Performance Measures,” *Weather Forecast.*, vol. 19, no. 6, pp. 1106–1114, 2004.

[101] J. M. Bland and D. G. Altman, “Measurement error.,” *BMJ*, vol. 312, no. 7047, p. 1654, 1996.

[102] A. K. Shrivas, “An Efficient Decision Tree Model for Classification of Attacks with Feature Selection,” vol. 84, no. 14, pp. 42–48, 2013.

[103] U. Fayyad and G. Piatetsky-Shapiro, “Advances in Knowledge Discovery and Data Mining,” 2004.