CONTENTS

LIST OF TABLES

LIST OF FIGURES

Rapport-Gratuit.com

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

LIST OF ABREVIATIONS

ANN	Artificial Neural Network
API	Application Programming Interface
APK	Android Package
ASLR	Address Space Layout Randomization
B-SIPS	Battery-Sensing Intrusion detection Protection System
CFG	Control Flow Graph
CIDE	Correlation Intrusion Detection Engine
CPU	Central Processing Unit
DAC	Discretionary Access Control
DT	Decision Tree
FN	False Negative
FP	False Positive
GID	Group Identifier
GMM	Gaussian Mixture Model
GPS	Global Positioning System
HMM	Hidden Markov Model
ICC-ID	Integrated Circuit Card Identifier
IMEI	International Mobile Equipment Identity
IMSI	International Mobile Subscriber Identity

IPC	Inter-Process Communication
K-NN	K- Nearest Neighbors
LR	Logistic Regression
MB	Mega Byte
MMS	Multimedia Messaging Service
NB	Naïve Bayes
OS	Operating System
RADS	Remote Anomaly Detection System
RAM	Random Access Memory
ROC	Receiver Operating Characteristics
SIM	Subscriber Identity Module
SMS	Short Message Service
SVM	Support Vector Machine
TN	True Negative
TP	True Positive
UID	User Identifier
WiFi	Wireless Fidelity
XML	eXtensible Markup Language

INTRODUCTION

Context of the Thesis

Over the last two decades, mobile phones have become cheaper, smaller, more sophisticated, stylish and “smarter” than ever before. Initially, mobile phones were of large size and had long antennas, short memory and lower battery life. The first commercial mobile phone was released by Motorola in 1983. It cost around $4,000 and had a thirty minutes battery life along with the capacity of storing only thirty contact numbers (Engineers’ Forum, 2013). Ever since, mobile phone manufacturers have been producing phones with new features and higher capacities. Twenty-four years after the first mobile phone was introduced, the first full touchscreen mobile phone was unveiled by Apple. This new model was called iPhone, a new generation of mobile phone with innovative features. The most important of these innovations was third-party applications, which allow users to develop and run their own applications designed for specific tasks. In the next year, the first Android mobile phone was introduced, the T-Mobile G1 or HTC Dream, which had a slide-out keyboard and limited touchscreen. Android is an open source mobile operating system used for a wide range of mobile devices.

Figure 0.1 (Engineers’ Forum, 2013) summarizes the evolution of mobile phones. Mobile phones have evolved considerably in terms of their capacity, technology and functionality. Mobile phones today are as powerful as personal computers.

A smartphone is a mobile phone that provides advanced functions as compared to traditional mobile phones. Smartphone systems have the capability to take photos with a high resolution camera, play music, receive and send emails, play games, surf the web and run third-party applications, functions which were unimaginable in earlier versions of mobile phones. The importance and the use of smartphone systems continue to grow rapidly. According to the International Data Corporation (IDC), 305 million smartphone units were shipped in 2010 (IDC, 2011), and this number is expected to surpass 1.4 billion units in 2015, accounting for 69 % of all smart connected device shipments worldwide (IDC, 2013). Figure 0.2 illustrates the market share forecast for desktop PCs, portable PCs, tablets and smartphones.

With third-party applications, smartphone systems provide a wide range of functions. In the last three years, more than 300,000 applications have been developed. As specified by a new IDC forecast, the market of smartphone applications will continue to grow, as the annual mobile application downloads on smartphone systems, tablets, and similarly specified mobile computing devices will increase from 87.8 billion in 2013 to 187 billion in 2017. In addition, revenues will experience similar growth, as they are forecast to increase from $10.3 billion in 2013 to $25.2 billion in 2017 (John, 2013).

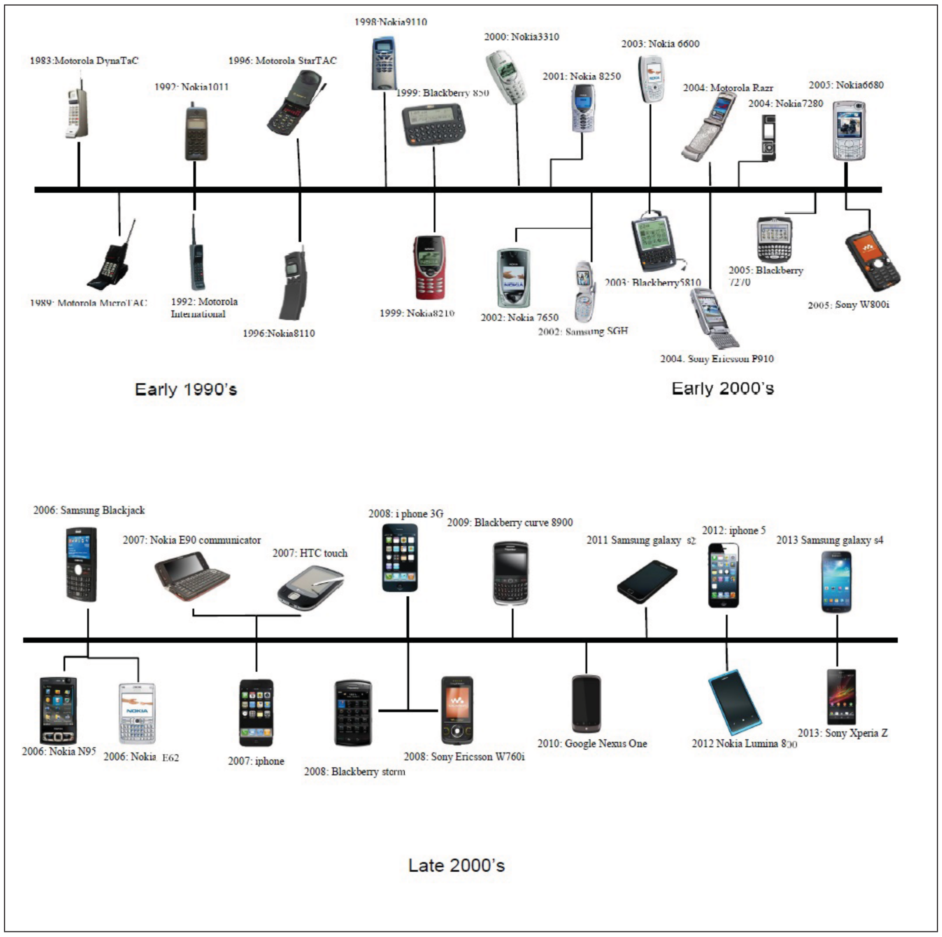


Figure 0.1 The evolution of mobile phones
Taken from (Engineers' Forum, 2013).

Smartphone systems have become so useful and important in personal and business life. Users can store all kinds of confidential and important information on their devices, such as credit card and bank account information. The popularity and the confidential information are two attractive things for cyber criminals and malware developers. In the last decade, smartphone systems have become an important target of malware developers.

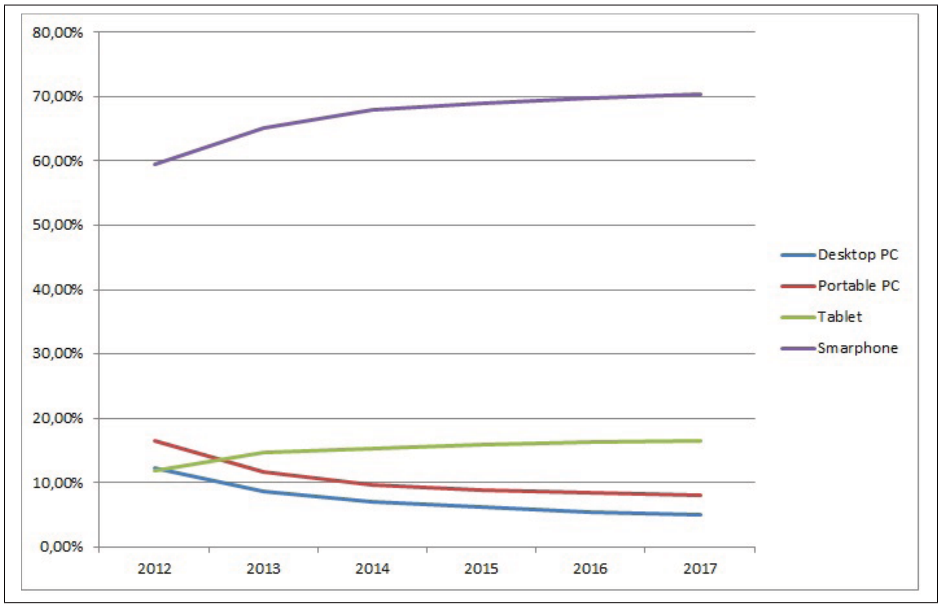


Figure 0.2 Worldwide smart connected device forcast (IDC, 2013).

The story of smartphone malwares started in 2004. In that year, security researchers made the first smartphone worm named *Cabir* (CNCCS, 2010; Chen and Peikari, 2008). It is a worm that was developed as proof of concept to infect Symbian-based devices and spread via Bluetooth as a *.sis package. In the same year, the Trojan *Qdial* (CNCCS, 2010; Chen and Peikari, 2008) was found as cracked copy of Symbian game, Mosquitos. This malware would send SMS massage to premium rates services, making it the first malware to take money from its victims.

By 2005, smartphone malware was declared as an information theft. *Pbstealer* (CNCCS, 2010; Chen and Peikari, 2008) is a malware that copies all information from an address book and then tries to send them via Bluetooth. *Commwarrior* (CNCCS, 2010; Chen and Peikari, 2008) is the first malware to spread via MMS, instead of Bluetooth. It is a harmless malware, but represents a major step in the smartphone malwares evolution. By early 2006, smartphone malware was targeting platforms other than Symbian, such as Windows CE and J2ME.

By 2010, the first Android trojan named *AndroidOS.DroidSMS.A* (Passeri, 2011) was introduced, which was a Russian SMS fraud application. In the same year, another trojan named *DROIDSMS.A* (Passeri, 2011) was discovered in the game TapSnake, which would transmit the GPS location of the infected devices. Android has become the most targeted mobile platform.

Smartphone malwares keep growing both technologically and structurally. They are increasingly becoming smarter and harder to detect, as illustrated in Table 0.1. Moreover, their number keeps increasing every year. Kaspersky Lab published an analysis report of smartphone threats (Chebyshev and Unuchek, 2014). The report provides shocking information about smartphone threats such as:

- Android remains the most targeted platform by 98.05 % of malicious attacks detected in 2013.
- Approximately 143,000 new malwares for smartphone systems have been detected in 2013. The number is three times greater than the previous year.
- In 2013, the cybercriminals have used nearly 4 million installation packages to distribute malwares for smartphone systems.
- A total of 10 million unique malicious installation packages targeting smartphone systems have been detected overall 2012-2013.
- The majority of malwares in 2013 are used to steal users' money.

More and more users and businesses use smartphone systems as computing appliances for both private and work life. Smartphone malwares are real and reach far beyond simple abusive behavior. Therefore, the need for efficient smartphone malwares detection solution is now crucial. This solution should be adequate with available system resources and should have as

Table 0.1 Smartphone malwares capabilities evolution.

Smartphone malwares capabilities	**2004**	**2014**
Spread via Bluetooth	Yes	Yes
Show messages from cybercriminals	No	Yes
Send SMS	No	Yes
Intercept SMS	No	Yes
Steal contacts	No	Yes
Steal banking data	No	Yes
Steal control bank account	No	Yes
Steal stored files	No	Yes
Steal data from other apps	No	Yes
Record phone calls	No	Yes
Redirect or block phone calls	No	Yes
Block device	No	Yes
Erase all data	No	Yes

high performance as possible in terms of security protection while consuming as less system resources as possible.

Problem statement

In the last few years, smartphone malware detection techniques have been actively studied. Prior work has shown that the two main adopted techniques were: *signature-base techniques* and *anomaly-based techniques* (Amamra *et al.*, 2012b). Each technique has its own strengths and drawbacks.

Signature-based techniques model the known malwares in the form of signatures or patterns, and use these signatures in the detection process (Amamra *et al.*, 2012b). All signatures are stored in a database. This database is searched when the detector attempts to identify whether a program is malicious or not. To keep the malware detector accurate and efficient, the signature database must be updated as soon as new signatures are available. Generally, malware signatures are sequence of bytecode (Rastogi *et al.*, 2013) and/or hash values (Rastogi *et al.*, 2013; Cha *et al.*, 2011).

Rapport-Gratuit.com

Signature-based techniques dominate the commercial malware detection solutions. They are very efficient and reliable to identify known malwares. However, they suffer from major drawbacks:

- They cannot detect unknown malwares and variants of known malwares (Amamra *et al.*, 2012b).
- They are easily evaded by using different techniques, such as binary packers, encryption, or self-modifying code, which means recycling existence malware with different signatures (Rastogi *et al.*, 2013; Christodorescu and Jha, 2006; Idika and Mathur, 2007).
- The signature extraction needs human expertise intervention, which is time consuming. The average time required for signature-based antivirus solution to detect new malware is 48 days (Schmidt, 2011).
- The malware developers are increasingly using complex obfuscation techniques, which make the code analysis more difficult, along with making signature extraction harder and time consuming. Similarly, the antivirus requires more time to detect the malware (Chebyshev and Unuchek, 2014).
- Zhou and Jiang (2012) evaluated four popular antivirus software with 1260 malware samples (Zhou and Jiang, 2012). The highest and lowest accuracy rates that were found are illustrated in Table 0.2. The results show clearly that the commercial antiviruses need more improvement to ensure the smartphone system protection.

Table 0.2 Commercial antivirus malware detection rate
Taken from (Zhou and Jiang, 2012).

AVG		**Lookout**		**Norton**		**Trend Micro**	
Nb	%	Nb	%	Nb	%	Nb	%
689	54,70%	1003	79,60%	254	20,20%	966	76,70%

Anomaly-based detection techniques are more interesting from the perspective of evasion attacks, obfuscation techniques and detection of unknown malwares. Instead of malware signatures database, anomaly-based detection techniques broadly construct benign behavior profiles during the training phase. In the detection phase, any deviations from these profiles are considered anomalous. These two phases are illustrated in Figure 0.3.

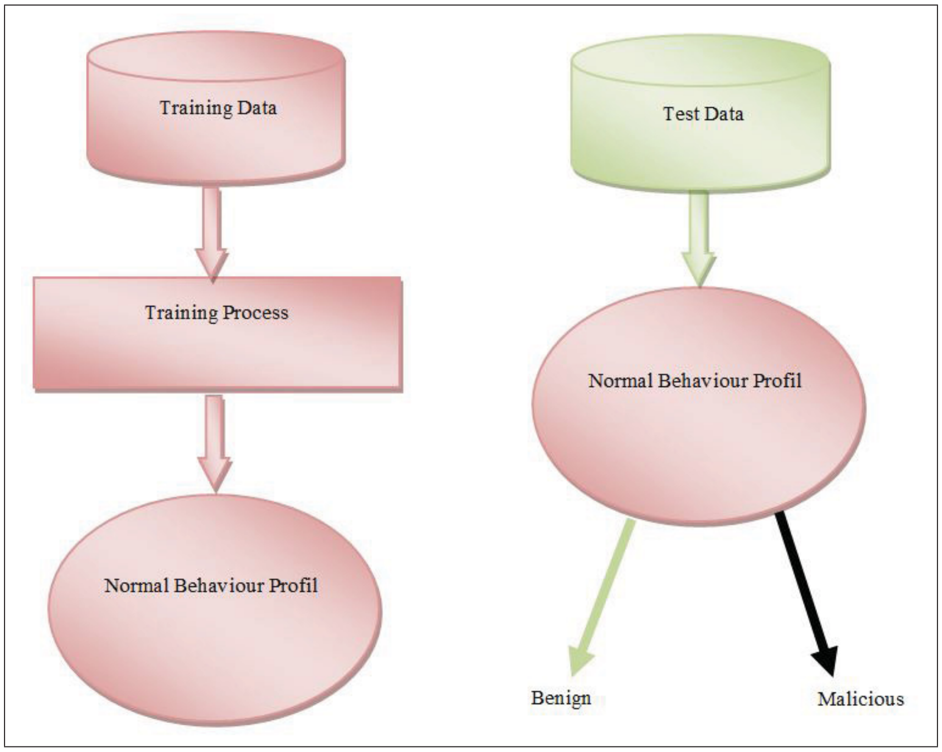


Figure 0.3 The phases of anomaly-based techniques.

In order, to be effective, anomaly-based techniques must have a stable and consistent benign behavior profiles. Constructing these profiles is not simple, especially for complex programs. Inadequate profiles of benign behavior lead to low accuracy and high false positive rate (Amamra *et al.*, 2012b).

To construct robust benign behavior profiles, different approaches such as Hidden Markov Model (HMM) (Xie *et al.*, 2010), Support Vector Machine (SVM) (Bose *et al.*, 2008), and k-means (Burguera *et al.*, 2011) have been implemented. These algorithms have been used on different datasets: system calls (Burguera *et al.*, 2011), system resource usage (Shabtai *et al.*,

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

2012), requested permission (Sanz *et al.*, 2013; Aung and Zaw, 2013), API calls (Aafer *et al.*, 2013; Kim *et al.*, 2012).

These approaches usually produce accurate benign behavior profiles with low rates of false positive alarms. Unfortunately, this generally comes at expensive price of computational and space complexities. These complexities represent an important burden for limited resource environments such as smartphone systems.

In spite of efforts made by researchers to produce efficient smartphone system anomaly-based solution, there are still two major problems:

- Anomaly-based detection solution for smartphone system still needs improvements to increase the detection rate of malwares. The solution should effectively detect all types of malwares (new malwares, known malwares, variant of known malwares) with very low false positive rate. The poor accuracy performance is caused by different factors; such as the used data is unrepresentative of the application behavior, the amount of used data is insufficient and/or using inappropriate benign behavior model.
- Smartphone hardware industry keeps its evolution of improving mobile device resources (increase memory size and CPU performance), but at the same time, mobile software keeps growing and requires more memory space and computational time. Therefore, anomaly-based detection solution for smartphone system should be adequate with available system resources and consume as less system resources as possible.

Thus, in this Thesis, we address these two issues to reach the objectives mentioned the next section.

Objectives and contributions

Our primary goal is to revisit the anomaly-based detection techniques based on system call dataset with two conflicting objectives in mind:

- Augment the accuracy, efficiency and adaptability of the solutions.
- Reduce the time and space complexities of the solutions.

To tackle these objectives, this Thesis presents contributions mainly on three levels:

a. **Prior work study level**

   We carefully studied and reviewed the existing smartphone malware detection techniques. We provided a structured and comprehensive classification of those techniques according to well defined and structured criteria: the reference behavior, the analysis techniques, the algorithms and the dataset used to construct the model. This survey helps us to understand the current trends in smartphone malware detection techniques, identify the advantages and the drawbacks of each technique, and explore possible new trends and improvements.

b. **Dataset level**

   To improve the detection accuracy of malwares, earlier work investigated only two factors: first, the used dataset (size and type) and second, the used algorithm to model the benign behavior. Our contribution on dataset level is twofold:

   - Firstly, we introduce and investigate *feature vector representation* factor. feature vector is the format of how data is organized and represented. This format is used during the training phase of the classifier as well as the detection phase. The investigated feature vectors are:
     - Successive system calls, where the ordering information between system calls in sequence is considered. This feature vector has been used in prior work.
     - Bag of system calls, where the successiveness of system calls in sequence is disregarded, and only the frequency of each system call is preserved. This feature vector has been used in prior work.
     - Patterns frequency system calls feature vector combines features of two previous vectors. Pattern-frequency feature vector regards the successive order information of system calls in short pattern, and regards the frequency of each pattern in the

sequence. To the best of our knowledge, this representation has not been studied in the past in this context.

The process of preparing feature vectors has light computational and space complexities which make it a suitable approach for smartphone malwares detection techniques.

- Secondly, we introduce a process named the *filtering and abstraction process*. A similar idea has been used in the past in different contexts (Wang *et al.*, 2009). This process refines the system call traces and filters out system calls that are irrelevant to describe the main behavior of applications, such as system calls to maintain process information, system calls to check resources availability, inter-process communication system calls, memory management system calls and failed system calls. This process also unifies system calls having the same functionality but using different names. For example, the system calls *read()*, *readv()*, *pread()* and *fread()* have similar function, read data from a file are considered as one system call.

The filtering and abstraction process produces refined traces. These traces have two main advantages over the raw traces:

- The refined traces are much more compact. This reduces the resources needed to store and process these traces for anomaly detection.
- The refined traces should give a better description of the behavior of applications. This description is expected to have positive impact on the classification accuracy of the anomaly-based detection approach.

c. **Benign behavior model level**

Benign behavior model is an important factor that impacts the accuracy performance of the anomaly-based detection technique. An inappropriate benign behavior model increases the false positive rate. Therefore, our contribution on benign behavior model level is twofold:

- Firstly, we adopt and extend the classical lightweight anomaly detection approach proposed by (Forrest *et al.*, 1996). The benign behavior of a legitimate application can be described by a database composed of all the unique system call patterns of a given length *k* encountered during a training phase (Forrest *et al.*, 2008; Hofmeyr *et al.*, 1998). Unfortunately, there are millions of applications available for Android system. It is unlikely that all the legitimate applications would have their authenticated system call database describing their benign behavior. Hence, our contribution is to build a canonical database representing generic benign behavior of Android applications. Our approach is to build this database from a limited number of representative applications. If the behavior of any application deviates from the behavior described by the canonical database, it should be reported as a suspicious/malicious application. In an in-depth protection strategy, this may represent the last opportunity to detect any malicious activity of a given application.

- Secondly, the prior work investigated and evaluated benign behavior model that produced by training only single machine learning classifiers, such as support vector machine (SVM) (Bose *et al.*, 2008; Zhao *et al.*, 2011), decision tree and Naïve Bayes (Shabtai *et al.*, 2012) for smartphone malwares detection problem. Therefore, we enhance the accuracy of malware detection process by modeling the benign behavior by training hybrid machine learning classifiers. A hybrid classifier combines two or more different single classifiers to improve classification process performance and reduce false positive rate. Thereby, hybrid classifier is more accurate and better candidate to make accurate malware detector. Stacked generalization or stacking is proposed by Wolpert (1992) is used to combine the two single classifiers. The main idea of stacking approach is to combine multiple classifiers by training the meta-level classifier by the output predictions of the base-level classifier, this allows correcting the misclassified instances and improving the prediction accuracy.

Methodology

To achieve the objectives and the contributions of Thesis, the following methodology is adapted:

- We study the Android security mechanisms and architecture and identify the strengths and weaknesses of the current mechanisms. As well as, we give a special attention to identify the possible alternatives for integrating a new mechanism that can improve the performance of the current mechanisms.
- We review smartphone malware detection systems and identify their weaknesses. Then, we explore the opportunities for improvement. Complexity analysis methods are used to compare different mechanisms according to their memory and time complexities.
- We propose a new malware detection system that is based on anomaly detection techniques. The new detection system is host-based since all malwares get installed on the device with the approval of users themselves. However, the smartphone resources are limited, the new detection system should be efficient and fit the available resources.
- We test and evaluate the proposed detection system on real benign and malicious applications. We evaluate its accuracy performances as well as the system resources consumption.

Structure of Thesis

This Thesis is organized in introduction, six chapters and conclusion as follows:

- The introduction explains the context of the Thesis, presents the problem statement and our intended objectives and contributions.
- Chapter 1 provides the basic background of Android smartphone system, Android security mechanism and their limitations and presents an overview of smartphone system security threats.
- Chapter 2 presents the relevant state-of-the-art techniques, methods, algorithms, and dataset related to the smartphone anomaly malware detections, along with their advantages and drawbacks.
- Chapter 3 describes Linux system calls and the importance and the limitations of using it as dataset in anomaly detection problem. As well as, we introduce our contributions on dataset

level, which presented in the different feature vector representations and the filtering and abstraction process.

- Chapter 4 presents the canonical benign behavior database approach. In this approach, the benign behavior of Android application is presented in a canonical database constructed from a limited number of representative applications.
- Chapter 5 compares between the two main categories of machine learning classifiers: generative classifiers and discriminative classifiers by choosing classifier from each category. k-means is selected to represent the generative category and support vector machine (SVM) is selected to represent the discriminative category. In this chapter, we examine the impact of filtering and abstraction process on the performance and accuracy of the two classifiers. The hybrid classifiers using stacking approach is presented in chapter 6.
- Finally, in our conclusion, we summarize the work accomplished during the Thesis, the contributions and prospects for the future.

Rapport-Gratuit.com

CHAPTER 1

BACKGOUND: ANDROID SECURITY MECHANISM AND THREATS

1.1 Android software stack

Android dominates smartphone operating systems market nearly 85% of the market share in the second quarter of 2014 (IDC, 2014). Android is an open-source software stack for a wide range of smartphone systems and tablets. It incorporates an operating system, middleware, and key applications. It consists of 5 layers as illustrated in Figure 1.1 (source.android.com, 2014; Gandhewar and Sheikh, 2010; Song *et al.*, 2010; Shabtai *et al.*, 2009a). The top layer is the application layer for implementing native and third-party applications, such as phone application, internet browser and media player application.

Android applications directly interact with classes of the application framework layer. These classes manage the basic functions of smartphone system. The most important classes are: (i) Activity manager manages the activity life cycle of applications, (ii) Content providers manage the data sharing between applications, (ii) Telephone manager manages all voice calls, (iv) Location manager manages locations and (v) Resource manager manages various types of resources (source.android.com, 2014; Gandhewar and Sheikh, 2010; Song *et al.*, 2010; Shabtai *et al.*, 2009a).

The next layer in Android software stack is native libraries. All libraries are written in C/C++, and they enable the smartphone system to handle various types of data. For instance, playback and recording different video and audio formats are supported by media framework library, *Libc* supports system C libraries, SGL and OpenGL are graphics library to support 2D and 3D graphics, and SQLite is a database engine (source.android.com, 2014; Gandhewar and Sheikh, 2010; Song *et al.*, 2010; Shabtai *et al.*, 2009a).

Android runtime layer is located on the same level as the libraries layer. It includes core libraries and *Dalvik virtual machine*. Core libraries provide most functions of java libraries and additional Android specific functions. Dalvik VM is a register based virtual machine optimized to execute applications of (.dex) format on limited resources devices. The Dalvik VM relies on Linux kernel for underlying functionalities, such as threading and low-level memory management (source.android.com, 2014; Gandhewar and Sheikh, 2010; Song *et al.*, 2010; Shabtai *et al.*, 2009a).

Android is based on Linux kernel version 2.6. Linux kernel is an abstraction layer between the hardware and the rest layers of the software stack. It is responsible for device drivers, such as camera and GPS, power management, memory management, security and networking (source.android.com, 2014; Gandhewar and Sheikh, 2010; Song *et al.*, 2010).

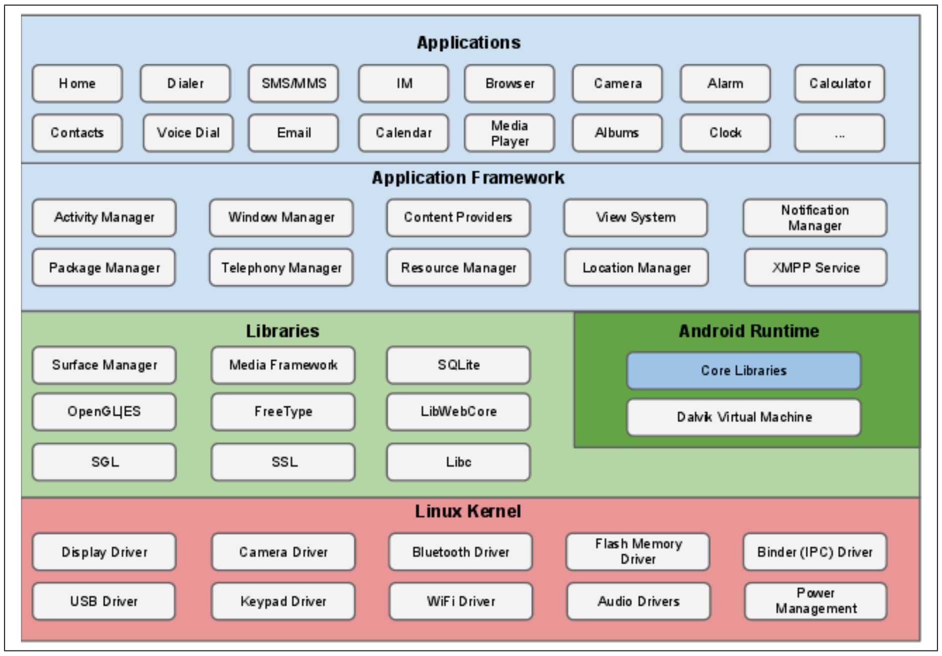


Figure 1.1 Android software stack architecture
Taken from (source.android.com, 2014).

1.2 Android security mechanism overview

Securing an open-source software stack requires robust and rigorous security mechanism. Android was designed with multi-layered security mechanism, which provides the flexibility required for an open-source platform, while providing protection for user data and system resources, such as networking and storage (Shabtai *et al.*, 2009a; source.android.com, 2014).

Android security mechanism broadly relies on two main security mechanisms: Linux security mechanism and Android application security mechanism, which are presented in the next subsections.

1.2.1 Linux security

Linux kernel is stable and secure kernel, trusted by many corporations and security professionals. Linux kernel provides Android with various security features including: process isolation, access permission model and extensible mechanism for memory protection (source.android.com, 2014; Shabtai *et al.*, 2009a).

1.2.1.1 Sandboxes

Android inherits Linux sandbox feature to enforce inter-application separation. Each Android application is executed in a separate process. The Android system assigns a unique user ID (UID) to each installed application (source.android.com, 2014; Shabtai *et al.*, 2009a). Sandbox feature isolates applications from each other and from the system. Consequently, each application data and resources are prevented from being used by other applications. Conceptually, a sandbox is illustrated in Figure 1.2.

In order for two applications to be executed in the same process, they must be signed by the same private key and share the same UID by declaring *sharedUserId* attribute in the AndroidManifest.xml file of each application. By doing this, the two applications are then treated as being the same application.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

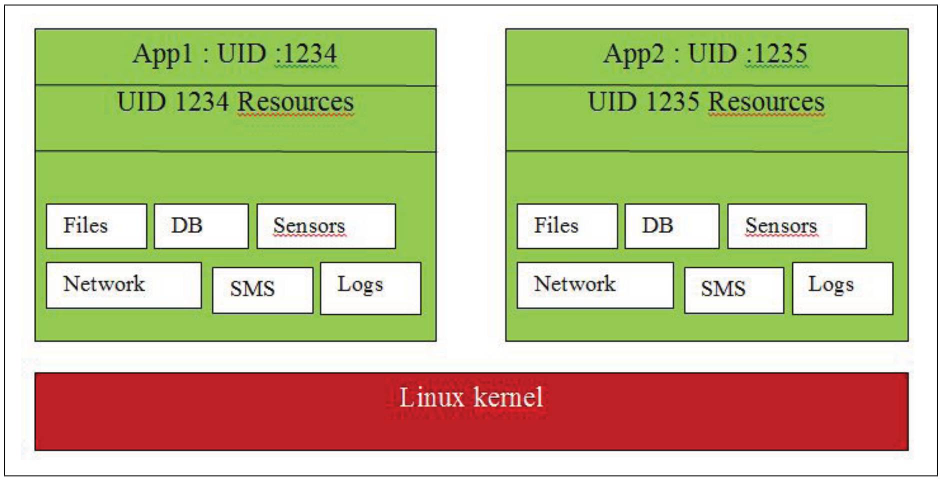


Figure 1.2 Linux sandbox security feature.

1.2.1.2 File access

Android relies on Linux discretionary access control (DAC) (Smalley and Craig, 2013; Shabtai *et al.*, 2009a) to protect both applications and system files. Each file is associated with user and group identifier (UID and GID, respectively) and three tuples read, write and execute permissions. Generally, Android system files are owned by superuser or root user, and application files are owned by application user ID (UID). This mechanism prevents one application from directly accessing or altering the files of the system or files of other applications through kernel interfaces.

1.2.1.3 Memory management

Android is enhanced with many memory management features (source.android.com, 2014) that make common memory corruption issues harder to exploit, including:

- Hardware-based No eXecute (NX) to deny code execution on the stack and heap.
- Linux *mmap_min_addr* to mitigate null pointer dereference privilege escalation.
- Address space layout randomization (ASLR) and ProPolice to protect from buffer over flow attack.

- Format string vulnerability protections.

1.2.2 Android application security

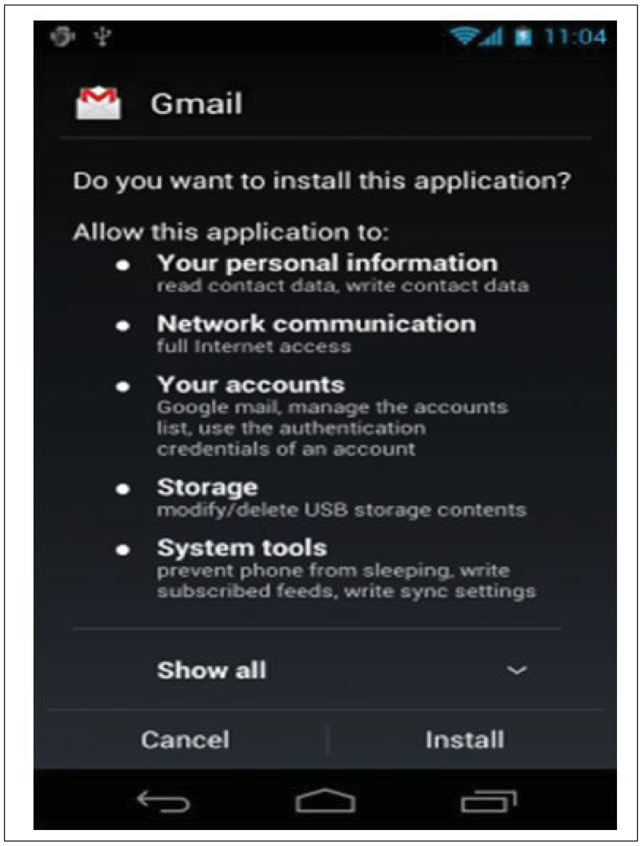
Additional finer-grained security features are provided at application level to enforce more Android security mechanisms. The two main security features are: application permission and application signing (source.android.com, 2014). First, by default, Android applications cannot access sensitive system resources, such as camera, GPS and network. These functions are protected by security feature known as permissions. Second, all Android applications must be signed digitally by a private key. The signing process identifies application's developer and detect if the application code altered. These two features are detailed in the following text.

1.2.2.1 Application permission

Third-party applications can access various system resources and data through a set of APIs. By default, Android applications cannot use the sensitive APIs that access network, device sensors, reading or writing private information, such as address book, or access other applications files. To use these sensitive APIs, Android application must explicitly declare the required permissions in its AndroidManifest.xml file (source.android.com, 2014; developer.android.com, 2014a).

Applications statically declare the permissions they need using <*uses-permission*> tags in AndroidManifestfile.xml. At the time of application's installation, Android system displays the requested permissions to the user, and asks whether to accept and continue the installation, or to deny and cancel installation as presented in Figure 1.3. User cannot accept some permissions and deny others. User must accept or deny all requested permission as a block.

Android system adopts the approach of showing the requested permissions prior to installation rather than other approaches for many reasons. This approach allows user to review information about application, developer and functions before establishing a mental of financial commitment, as well as to determine whether it corresponds to their needs and expectations.


11:04

Gmail

Do you want to install this application?

Allow this application to:

- **Your personal information**
  read contact data, write contact data
- **Network communication**
  full Internet access
- **Your accounts**
  Google mail, manage the accounts list, use the authentication credentials of an account
- **Storage**
  modify/delete USB storage contents
- **System tools**
  prevent phone from sleeping, write subscribed feeds, write sync settings

Show all

Cancel Install

Figure 1.3 Display of requested permissions at application installation time.

The APIs that are protected with permissions can be clustered in four categories: cost-sensitive APIs, personal information APIs, sensitive data input devices and device metadata, as showed in Figure 1.4.

- Cost-sensitive APIs: this category represents any function that might charge a cost for the user or the network. The APIs used to access these functions are protected with permission security feature. These APIs include: Telephony, SMS/MMS, Network/Data and In-App Billing.
- Personal information APIs: this category groups the APIs that provide access to sensitive information of user, such as contacts, emails and calendar.

- Sensitive data input devices: Smartphones are equipped with sensitive data input devices, such as camera, GPS and microphone. The APIs set to access these functions are protected by permission security feature.

- Device metadata: broadly, information such operating system logs, user browsing history, phone number and hardware/ network identification are not sensitive. However, they can describe the user behavior and their preferences, as well as the system usage and performance. Therefore, the APIs used to access the information are protected by permission security feature.

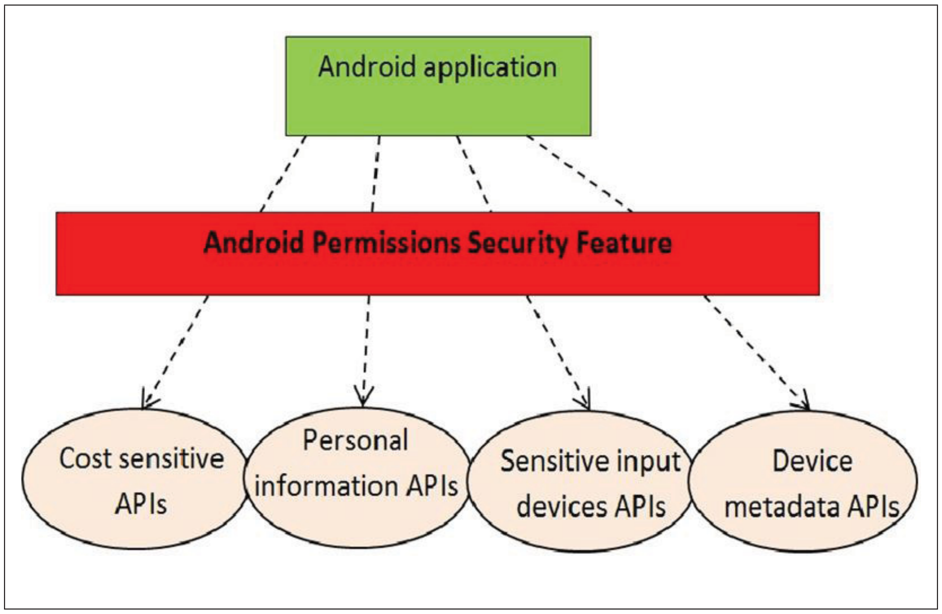

Android application
Android Permissions Security Feature
Cost sensitive APIs
Personal information APIs
Sensitive input devices APIs
Device metadata APIs

Figure 1.4 Classes of sensitive protected APIs.

1.2.2.2 Application signing

Each Android application must be digitally signed with a certificate whose private key is owned by the developer. Google play (official Android applications market) and the system will not accept an application that is not signed. The purpose of application singing is: (i) to identify the application's developer, (ii) to prevent application code and/or contents from altering and

(iii) to establish trust between applications to share code and data in a secure manner (developer.android.com, 2014a; source.android.com, 2014).

Application signing in Android is simple and developers can use self-signed certificate. There is no need for a certificate authority. Android provides two modes for signing applications: debug mode and release mode. Debug mode is used when the developer is developing and testing an application. While release mode is used when the developer wants to build and release version of an application that can be distributed to be used or published on an application marketplace, such as Google Play (developer.android.com, 2014a).

1.3 Android security mechanism limitations

Android security mechanism is varied and provides different capabilities of protection and prevention. However, some security limitations exist that will be discussed in this section (Shabtai *et al.*, 2009a) as follow:

- Users cannot accept only subsets of the permissions required by an application, it is an *"all or nothing"* policy. Indeed, a user should accept all the required permissions to allow the installation of a given application. These permissions are supposed to represent the minimum set of services without which the application cannot perform the tasks it was created for. However, in many cases, application developers exaggerate in requiring permissions, usually greater numbers of permissions are not really needed. For example, a weather forecast application requests permission to send an SMS, record audio, add or modify personal information, and directly call phone numbers. All those permissions have no relation with normal purposes of the application.

- There is no way to verify that an application is using the granted permissions only for benign purposes. In addition, the current security mechanism does not prevent the application from using granted permissions for malicious purposes, nor does it notify the user when an abuse is detected.

- During the installation process, when prompted to check the permissions requested by a new application, most of the users, unaware of security risks, tend to click “Yes, Ok, Accept, Install, etc.”. Later, the consequences of any bad use of the granted permissions fall on the shoulders of users.

- There is no dynamic way to constraint or revoke subset of permissions, the only way to revoke the application permissions is to uninstall the application.

- Android does not have any runtime investigation mechanism for the application’s behavior. Once an application is installed, it can do whatever it wants.

1.4 Smartphone system security threats

There are various security threats that can threaten smartphone system. These threats can be broadly classified into four classes (Dunham, 2008; lookout mobile security report, 2011; Zhou and Jiang, 2012): application-base threats, web-based threats, network-based threats and physical threats. These classes are discussed in more details in the next subsections.

1.4.1 Application based threats

Installing third party applications presents a security threat to mobile platform. There are applications developed with malicious tends, and other applications can be exploited for malicious purposes (Dunham, 2008; lookout mobile security report, 2011; Zhou and Jiang, 2012). Application-based threats can be one of the following forms:

- Malware is an application that is developed to execute malicious behavior on a smartphone system. For example, a malware performs actions without a user’s knowledge, such as making phone calls, sending SMS/MMS messages, or executing undesirable commands and programs to give an attacker remote control over the device. Malware can also steal sensitive information and send it outside the device, which leads to identity theft or financial fraud (Dunham, 2008; lookout mobile security report, 2011).

- Privacy threats caused by applications that are not necessarily malicious. Those applications gather or use sensitive information, such as GPS location, contact lists, phone calls activity, or personal identifiable information that are not necessary to perform their function or more than their needs (Dunham, 2008; lookout mobile security report, 2011).
- Threat caused by vulnerable applications that have not malicious behavior, but have vulnerabilities, which can be exploited for malicious purposes. Such vulnerabilities can allow the attacker to gain full access to a mobile device (Dunham, 2008; lookout mobile security report, 2011).

The aim of this Thesis is to propose a solution that tackle this type of threats. Therefore, malicious applications threats is explained in more details in Section 1.5.

1.4.2 Network based threats

Smartphone supports variety of wireless communications channels, cellular network, WiFi local network, Bluetooth as well. Every channel can be exploited by mobile malware to infect the handheld device.

- Bluetooth was the first channel targeted by smartphone malwares to infect systems and distribute themselves. Cabir was the first malware that used Bluetooth channel to infect smartphone systems (Dunham, 2008).
- Exploiting network software flaws such SMS/MMS is a serious security threat. (Mulliner and Miller, 2009; Mulliner and Vigna, 2006) identified new flaws in SMS and MMS software that can be used for Denial-of-Service attacks and code execution.
- WiFi sniffer captures wireless network traffic to or from the smartphone system. Many applications and web pages send their data through WiFi channel which are not encrypted so if the traffic is intercepted by anyone, this may lead to identity theft or financial fraud (Dunham, 2008).

1.4.3 Web based threats

Smartphone system constantly access Internet to check general web sites, read emails, Facebook or Twitter. Web services threaten smartphone system security in different ways as follows:

- Phishing is a way to steal private and sensitive information such as usernames, passwords, account numbers, social security numbers and credit card information by masquerading as a trustworthy entity in an electronic communication. Fishers send out an email, message text, Facebook or Twitter message that appears to be coming from a well-known and trusted sender. The message contains a link which directs victims to enter private and financial information at a spoofed website which appears identical to the legitimate one (Dunham, 2008).
- Web pages automatically start downloading a program, sometimes the user is prompted to run the downloaded program, while in other times the program runs automatically (Dunham, 2008; lookout mobile security report, 2011).
- Application exploit vulnerabilities in a web browser or any program that can be launched via a web browser such as a flash player, PDF reader, or image viewer. Simply by visiting a web page, an attacker can trigger a program to exploit browser vulnerabilities and install malware or perform other malicious actions on a device (Dunham, 2008; lookout mobile security report, 2011).

1.4.4 Physical threats

Smartphone is portable and contains sensitive and private user information. Therefore its physical security is important. Lost or stolen mobile device constitutes a serious security threat (lookout mobile security report, 2011).

Rapport-Gratuit.com

1.5 Malware threats

The term malware comes from two words: “malicious” and “software”. Malware is a malicious code developed to disrupt system operation, access unauthorized system resources, execute undesirable commands and programs, collect sensitive information and send it outside the device, and other abusive behavior. Malware will also seek to exploit existing vulnerabilities on systems making their entry quiet and easy (Chen and Peikari, 2008).

Malwares targeting smartphone systems are clearly on the rise. According to Kaspersky lab (Chebyshev and Unuchek, 2014), nowadays, These malwares are beyond than frivolity of individual hacker, but serious industry involves various types of actors: malware writers, testers, interface designers of both malwares and web pages that are used to distribute them, owners of programs that spread the malwares, and the botnet owners.

The number of malwares targeting smartphone systems in 2013 was three times more than the previous year, nearly 145,000 new samples. Malwares of smartphone systems have increased quickly in short period of time. Over the course of 2011, Kaspersky Lap (Maslennikov, 2011) recorded 5,255 new variants of malwares and 178 new families. However, the malwares disclosed only in December 2011 are more than over the entire 2004-2010 period. Recently, Kaspersky Lap (Chebyshev and Unuchek, 2014) reported around 10 million malwares were detected between 2012 and 2013.

Malware authors remain concentrating on Android OS, where 98.10% of all detected malwares in 2013 targeted Android system (Chebyshev and Unuchek, 2014). This occured due to three factors: (i) the popularity of Android, (ii) the vulnerability of its architecture, and (iii) the fact Android gave application developers more facilities and freedom to release and upload application to marketplaces. Figure 1.5 illustrates malwares for different smartphone systems.

Since the detection of the first Android malware in August 2010, Android malware developments have made great innovation in their capabilities, distribution and how to avoid detection. *Ginmaster* is a torjan program disclosed in 2011 in China. This torjan is distributed through legitimate applications by injecting its malicious code. One year later, *Ginmaster* resists detections by using obfuscating class names, encrypting URL and C& C instruction. In 2013, the developers of *Ginmaster* used more complex and smarter obfuscation and encryption techniques (Chebyshev and Unuchek, 2014).

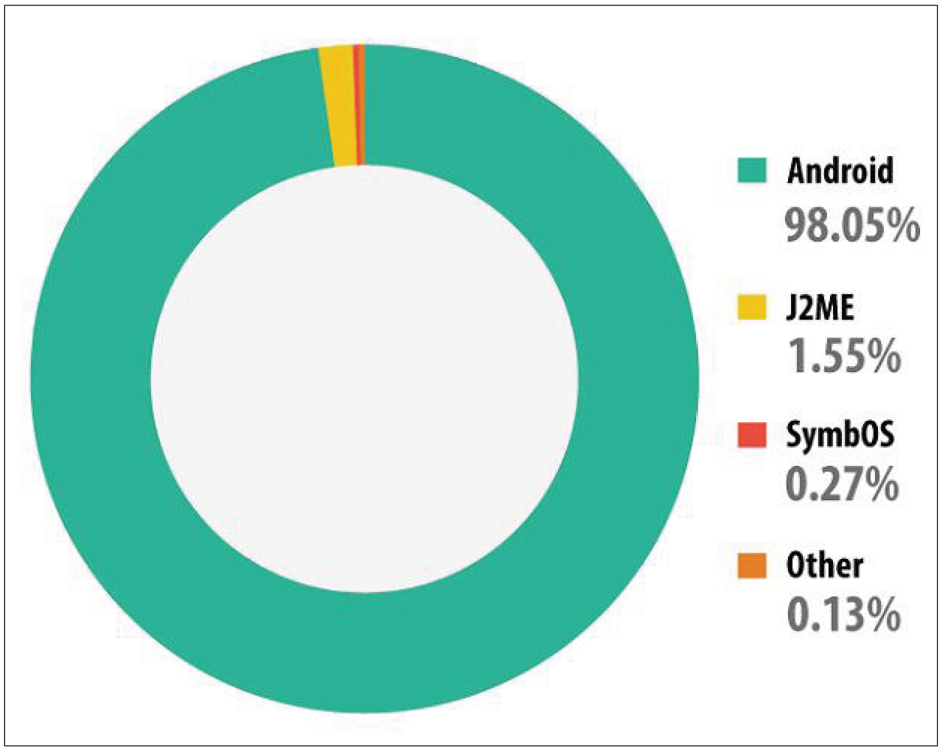


Figure 1.5 Smartphone malwares distribution in 2013 by OS
Taken from (Chebyshev and Unuchek, 2014).

1.5.1 Malware classification

The classification of smartphone malwares does not differ than computer malwares. The classification depends on their characteristics. These characteristics can be identified by answering precise questions such as:

- Is the malware stand-alone program or piece of code in a program?
- Is the malware self-replicating or non-replicating?

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

- Does the malware hide the malicious function or hide itself?

One possible classification of malware (Chen and Peikari, 2008) is presented in Figure 1.6.

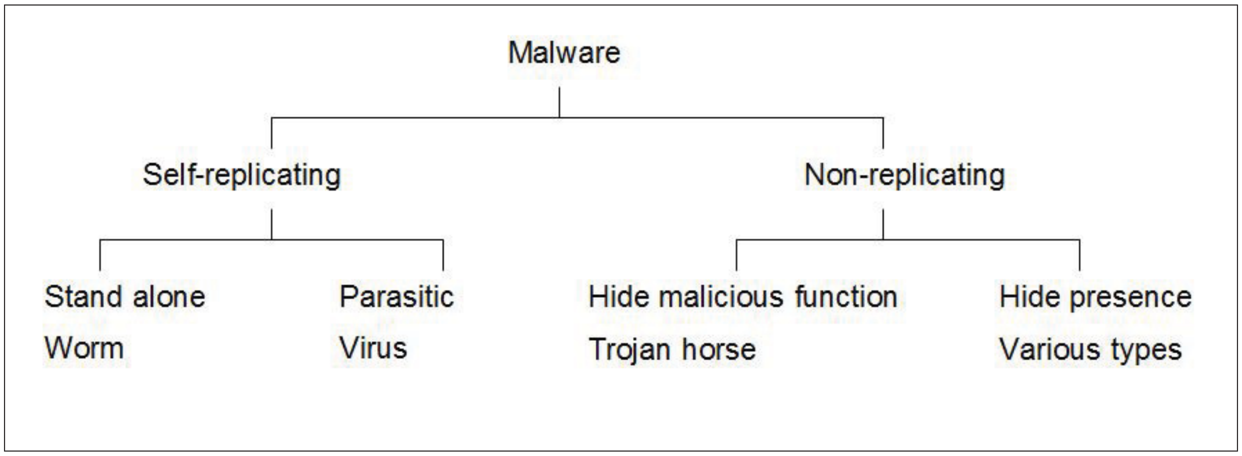


Figure 1.6 Smartphone malwares classification.

- A virus is a piece of software that attaches to an executable file, when the contaminated file is executing the viral code gets executed. A virus can overwrite programs on the system, destroy or delete data, use email to spread itself, or even erase everything on the memory. Virus is spread when the file is copied or moved from one system to another using the network, a disk, file sharing, or infected e-mail attachments (Chen and Peikari, 2008; cisco security, 2011).

- A worm is a stand-alone malicious program. It is self-replicating program, and uses communication channels to send copy of itself to other systems. Worm can delete files, encrypt files, send junk email, and consume network bandwidth (Chen and Peikari, 2008; cisco security, 2011). The first smartphone malware was worm, *Cabir*, which propagated through Bluetooth.

- A trojan horse is a program that presents itself as a legitimate program, while it actually has malicious activities. Trojan horse stays hidden on the infected system and achieves any types of attacks such as generating popping up windows, deleting files, stealing data, or activating and spreading other malware like viruse or bot. Trojan can open back door to give malicious users access to the system. Trojan horse is not self-replicating. It spreads by opening an email message attachment, copying files from storage media such as USB key, or downloading and running a file from the Internet (Chen and Peikari, 2008; cisco security, 2011). DroidKungFu is an example of Android Trojan that collects and transmits confidential data to remote server (threat description, 2011).

- A spyware is a program installed on the user device without his knowledge. The main tasks of spywares are: (i) monitoring user activities such as visited websites, received and sent emails, and exchanged instant messages, and (ii) collecting private information, such as usernames, passwords, account numbers, and even driver's license or social security numbers. The spyware transmits the collected information outside the infected system, usually for advertising purposes. Spyware is generally installed from a "free" or trail downloaded software. Some web sites will attempt to install spyware when you visit their page (Chen and Peikari, 2008; cisco security, 2011). Nickispy is an Android spyware that can records phone calls (Jiang, 2011).
- A botnet is a self-propagating malware that infects smartphone systems and connects back to a server that acts as a command and control (C&C) center for an entire network of compromised systems, or "botnet". A bot can register keystrokes, collect passwords and financial data, capture and analyze packets, launch DOS attacks, and open back doors on the infected device to provide unauthorized access to the device (cisco security, 2011).
- A trackware is a program that collects information that could be used to identify a user or device to an application. For example, an application that provides device location service (F-Secure, 2013).
- An Adware is any application in which advertisements are displayed during application execution in order to generate revenue for its developer. Adware has an additional code than benign application that displays the Ads. Unfortunately, some Adwares have intrusive behavior, such as leakage user's personal information or change system settings without the user's authorization (F-Secure, 2013).

The detected malwares are distributed over their classes as shown Figure 1.7 (Chebyshev and Unuchek, 2014). Smartphone malwares are clearly developed to steal user's money using SMS-Trojans (33.50%), Backdoors (20.60 %) and Trojans (19.40%).

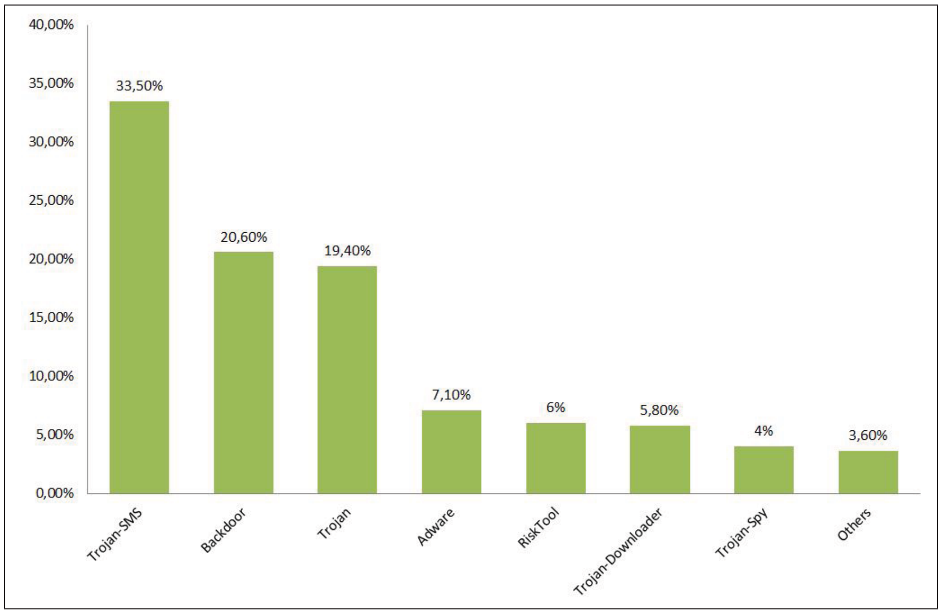


Figure 1.7   Smartphone malwares distribution by class.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

Due to sharp increase of malwares targeting smartphone systems, smartphone malware detection research field keeps attracting new researchers of different backgrounds. Smartphone malware detection is a new field of research. However, the number of published papers in this subject is constantly increasing and different approaches are proposed. The proposed approaches have been classified into different classes with different names such as: misuse techniques, dynamic techniques, anomaly techniques, behavior techniques, static techniques and signature-based techniques. Those classes are wide and general, where the same techniques are referred to in different research papers with different class's names. As far as we know, the literature has not proposed unified taxonomy with clear criteria to classify smartphone malwares detection techniques. Fine-grained and structured classification of smartphone malware detection techniques can greatly assist new researchers in this field to review the related literature easily, as well as where they should focus their research more on.

The aim of this chapter is to review the existing smartphone malware detection techniques and provide a structured and comprehensive classification of those techniques according to well defined and structured criteria. This classification should help us to understand the current trends in smartphone malware detection techniques and identify their advantages and drawbacks, as well as explore the possible future trends.

This study classifies smartphone malware detection techniques according to well defined criteria. Those criteria are inferred and complied from the existing malware detection researches (Chandola *et al.*, 2009; Idika and Mathur, 2007; Jacob *et al.*, 2008; Kabiri and Ghorbani, 2005; La Polla *et al.*, 2013; Vinod *et al.*, 2009) and can be summarized as follow:

- The reference behavior used by the malware detector to identify malicious and benign applications. The reference behavior can be benign behavior or malicious behavior.
- The analysis techniques used to analyse behavior of applications. The analysis techniques could be static or dynamic, and each technique extracts data that is used to represent application behavior.
- The algorithm used to model application's behavior.
- The data and information used to represent malicious behaviors.

According to these criteria, smartphone malware detection techniques could be classified into different classes. For example, reference behavior classifies the techniques to signature-based detection techniques that use malicious behaviors as reference behavior, and anomaly-based detection techniques that use benign behaviors as reference behavior. Analysis techniques classify the detection techniques to static or dynamic techniques. The malware detection techniques classification and their criteria are described in more details in the next sections.

2.2 Malware detection techniques classification criteria

A malware detector is a system responsible to determine whether a program has malicious behavior. In other words, malware detector D is defined as a function: $D : P \to \{malware, benign\}$ where P is a set of programs installed on the smartphone system, for a given program p

$$D(p) = \begin{cases} Malware & \text{if p contains malicious behavior,} \\ Benign & \text{otherwise} \end{cases} \tag{2.1}$$

Malware detectors use different algorithms, analysis techniques, data and approaches to detect malicious intents. Broadly, malware detection techniques for smartphone systems are classified into different categories according to well defined rules. These rules are inferred and compiled from literature reviews of malware detection techniques in general and recent contributions targeting smartphone malwares detection (Chandola *et al.*, 2009; Idika and Mathur, 2007; Jacob *et al.*, 2008; Kabiri and Ghorbani, 2005; La Polla *et al.*, 2013; Vinod *et al.*, 2009). Figure 2.1 gives a closer look to these rules and their possible categories.

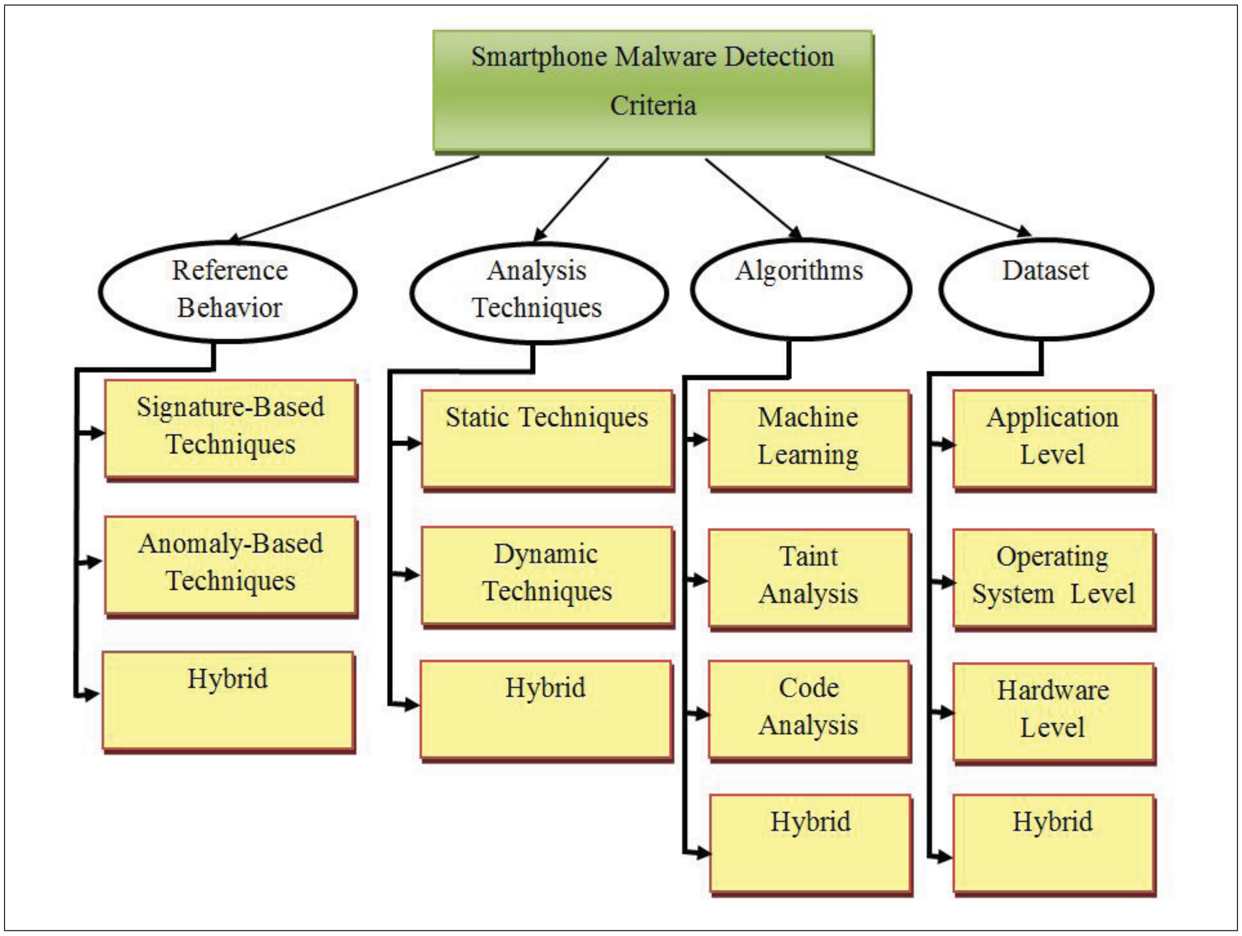


Figure 2.1 Overview of malware detection techniques criteria.

2.2.1 Reference behavior rule

Malware detection techniques use reference behavior to distinguish malicious applications from benign ones. This reference behavior can be malicious behavior, as in the case of signature-based techniques (referenced as misuse techniques in some references) or benign behavior, as in the case anomaly-based techniques (Idika and Mathur, 2007; Vinod *et al.*, 2009). The difference between the two techniques is illustrated in Figure 2.2. Signature-based detection techniques define every known malware by a signature or pattern. Then use the malware signatures to identify an application's behavior either malicious or benign. Anomaly-based detection techniques model benign behavior as reference behavior. Then use this benign model to identify an application's behavior either malicious or benign.

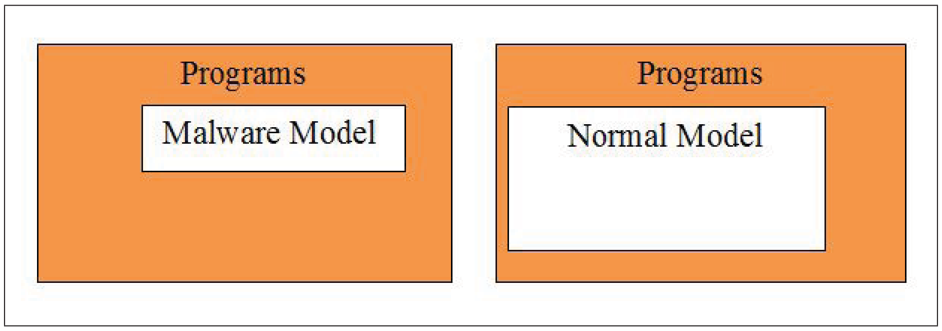


Figure 2.2 Reference behavior rule.

2.2.2 Analysis techniques rule

Analysis techniques are the techniques that used to study and understand the application's behavior, as well as extract data that can represent this behavior. The two main analysis techniques are: static analysis and dynamic analysis. Static analysis techniques study application's behavior from its code information without executing it. These techniques pass through different steps, such as unpacking, disassembling, analysing and extracting the data. Dynamic analysis techniques study application's behavior from its run-time information. These techniques install and run the application, then collect information, such as events, system call traces, and communication traffic. According to this rule, the malware detection techniques can be classified into two categories: static techniques and dynamic techniques (Amamra *et al.*, 2012b).

2.2.3 Algorithms rule

Various algorithms have been used for differentiating between malicious and benign applications, as well as improving the execution performance. Broadly, the most used algorithms are: (i) machine learning classifiers (Shabtai *et al.*, 2012; Amamra *et al.*, 2012c,b), (ii) taint analysis (Enck *et al.*, 2014), (iii) control flow analysis (Grace *et al.*, 2012), (iv) code inspection (Zhou *et al.*, 2012) and (v) multi-pattern matching algorithms (Amamra *et al.*, 2012a). Malware detection techniques can be classified according to the used algorithms.

2.2.4 Dataset rule

Different data have been studied and used to represent application's behavior. The used data is an influential factor on the detection accuracy and execution performance of malware detection techniques. The data can be extracted at the following levels: application, operating system (OS) and hardware. Broadly, the most used ones at application level are: hexadecimal bytes, hash values, and application code syntax and structural properties (Aafer *et al.*, 2013). At operating system level, run-time events and triggers such as system calls (Burguera *et al.*, 2011; Amamra *et al.*, 2012c). Finally, at the hardware level, the used data are the basic system metrics such as CPU usage, free or used memory and power consumption (Shabtai *et al.*, 2012).

2.3 Malware detection techniques classification

Figure 2.3 illustrates one possible classification of malware detection techniques using the different defined rules that are presented in the last section. Reference behavior rule classifies detection techniques broadly into two main categories: anomaly-based detection techniques and signature-based detection techniques. Anomaly-based detection techniques build benign behavior profile in training phase, and any deviation from this profile in detection phase is considered malicious. For each technique, a new rule can be used to split it into subcategories. For example, analysis techniques rule can classify anomaly-based detection techniques into static detection techniques or dynamic detection techniques. In static techniques, the program

Rapport-Gratuit.com

behavior is represented by static information such as, programs syntax, code instructions, APIs and program structural properties. While dynamic detection techniques use the run-time information that is collected from application's execution, such as system calls.

The malware signatures could be static signatures or behavior signatures. Static signatures are sequence of hexadecimal bytes or sequence of alphanumeric characters represents hash value. While behavior signatures are complex meta-structures embracing dynamic concepts and a semantic interpretation. Analysis techniques rule classifies behavior signature techniques into static behavior signature and dynamic behavior signature. In static behavior signature, the signature is extracted from code syntax and structural information. In dynamic behavior signature, the signature is extracted from code runtime information. Closer look to these techniques is presented in the next sections.

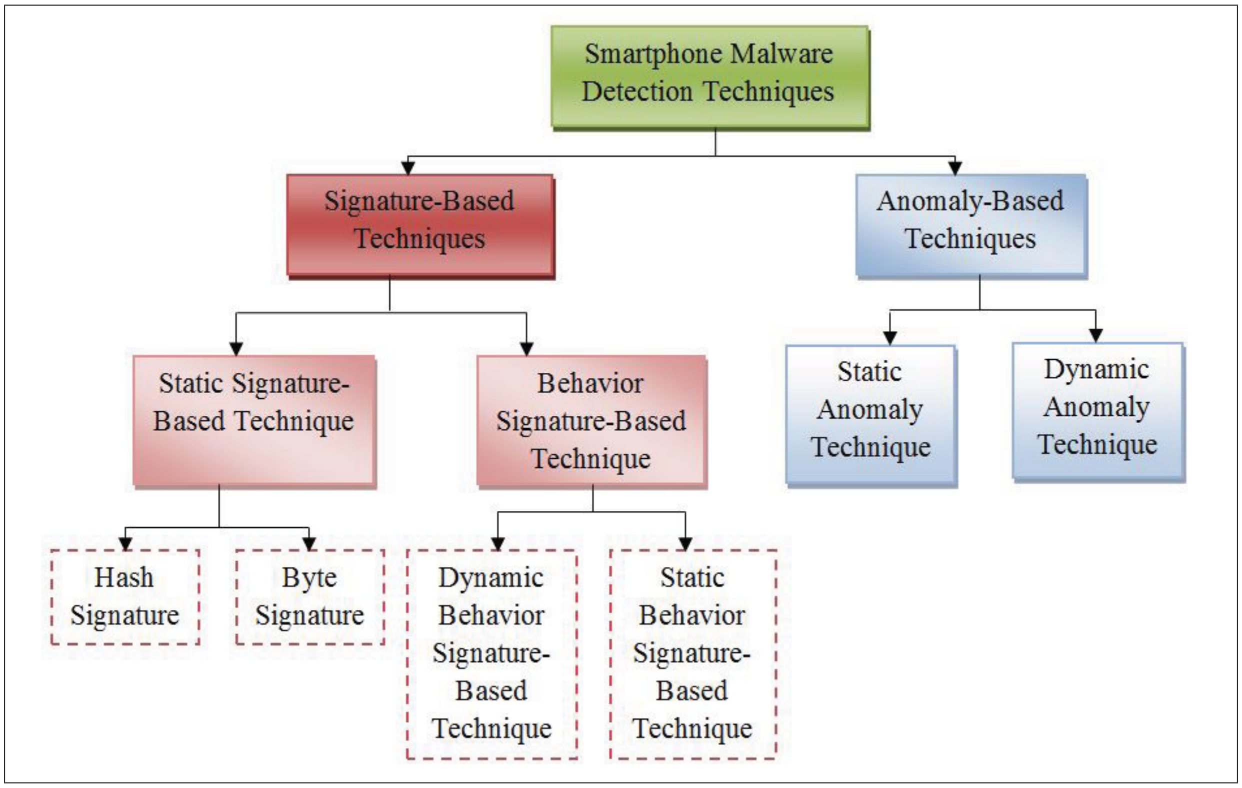

Smartphone Malware Detection Techniques
Signature-Based Techniques
Anomaly-Based Techniques
Static Signature-Based Technique
Behavior Signature-Based Technique
Static Anomaly Technique
Dynamic Anomaly Technique
Hash Signature
Byte Signature
Dynamic Behavior Signature-Based Technique
Static Behavior Signature-Based Technique

Figure 2.3 Smartphone malwares detection techniques classification.

2.4 Signature-based detection techniques

Signature-based detection techniques model the known malicious behavior of malware in form of signature and use this signature in the detection process. All malware signatures are stored in a repository. This repository represents malware signatures database and is searched when the detector attempts to identify whether or not a program contains known malicious behavior. To keep the malware detector accurate and efficient, the signature database must be updated as soon as a new signature is available. Signature creation needs human expertise, which introduces human error and considerable delay.

The malware signatures can be static signatures or behavior signatures. A static signatures are a sequence of hexadecimal bytes or hash values. The behavior signatures represent malware behaviors. According to the malware behavior representation rule, signature-based detection techniques are classified into static signature-based detection techniques and behavior signature-based detection techniques.

2.4.1 Static signatures-based detection techniques

Static signatures are adopted by the most commercial Antivirus. Some of them are presented in Table 2.1. Static signature-based detection techniques scan the phone RAM and SD card for patterns that match with one of its signature database. The most common signatures used in these techniques are byte-signatures and hash-signatures.

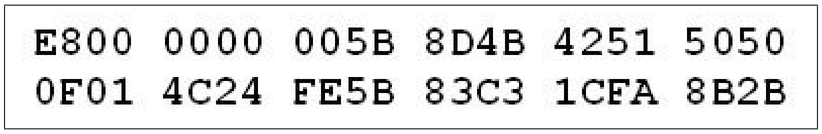
Table 2.1 Samples of commercial antivirus for smartphone.

Product	License	OS platform
Kaspersky Mobile Security	Commercial	Android, Symbian, Windows, Blackberry OS
Norton Mobile Security	Commercial	Android, Symbian, Windows, Blackberry OS
McAfee Mobile Security	Commercial	Android, Symbian, Windows, Blackberry OS
Lookout Mobile Security	Free	Android, iPhone OS
iCareMobile	Free	Android, Symbian

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

2.4.1.1 Byte code signature

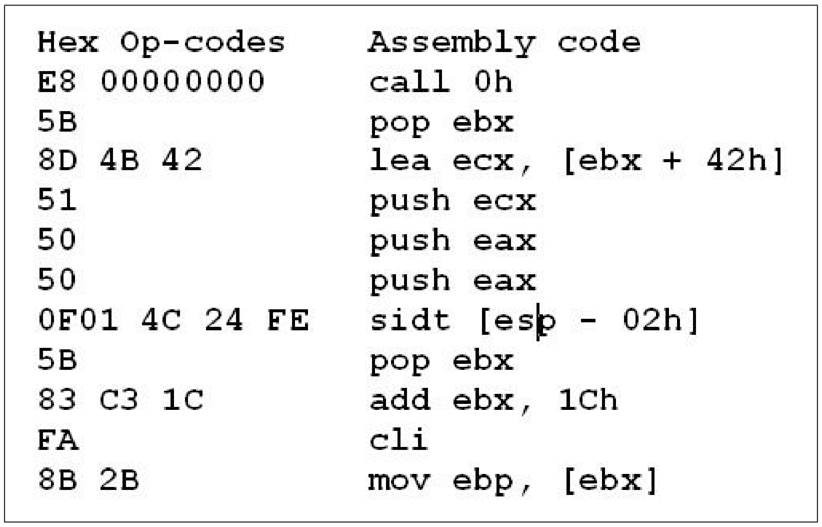
Byte code signature is sequence of hexadecimal bytes that are present in a file or data stream. It is very known form of detection and has been used since the first Antivirus detector. For example, the hexadecimal signature of the Chernobyl/CIH virus (Christodorescu and Jha, 2006) is illustrated in Figure 2.4.



```
E800 0000 005B 8D4B 4251 5050
0F01 4C24 FE5B 83C3 1CFA 8B2B
```

Figure 2.4 The hexadecimal signature of the Chernobyl/CIH virus.

The signature illustrated in Figure 2.4 corresponds to part of the virus body, and the IA-32 instruction sequence corresponding to the signature is presented in Figure 2.5.



```
Hex Op-codes      Assembly code
E8 00000000       call 0h
5B                pop ebx
8D 4B 42          lea ecx, [ebx + 42h]
51                push ecx
50                push eax
50                push eax
0F01 4C 24 FE     sidt [esp - 02h]
5B                pop ebx
83 C3 1C          add ebx, 1Ch
FA                cli
8B 2B             mov ebp, [ebx]
```

Figure 2.5 IA-32 instruction sequence corresponding to the Chernobyl/CIH virus.

Figure 2.6 illustrates the byte code signature of the famous Android malware, DroidDream (Rastogi *et al.*, 2013). The signature is given in smali (an intuitive assembly language for Dalvik bytecode).

```
const-string v10, "profile"
const-string v11, "mount -o remount rw system\nexit\n"
invoke-static {v10, v11}, Lcom/android/root/Setting;->
     runRootCommand(Ljava/lang/String;Ljava/lang/String;)
     Ljava/lang/String;
move-result-object v7
```

Figure 2.6   The byte code signature of DroidDream malware.

2.4.1.2   Hash signature

It is the most basic and easiest form of signature. Hash-signature is created by a hash function that converts data into sequence of alpha-numeric characters. The most commonly used hash functions are MD5 and SHA-1, where 84% of all signatures in ClamAV are MD5 (Cha *et al.*, 2011). For example, the MD5 hash values of two simple words are illustred in Figure 2.7.

```
MD5 (“Hello World”) = b10a8db164e0754105b7a99be72e3fe5
MD5 (“Hello World!”)= ed076287532e86365e841e92bfc50d8c
```

Figure 2.7   The MD5 hash of similar words.

The main drawback of hash-signature is that the hash function generates a new hash value if a byte changes in the same block. This leads to new signatures for the same malware. Table 2.2 presents some samples of famous Android malwares and their SHA-1 signature (Rastogi *et al.*, 2013).

Static signature-based detection techniques use multi-pattern matching algorithms (Commentz-Walter, 1979; Aho and Corasick, 1975; Wu *et al.*, 1994) to identify malicious instances. Therefore, optimising these algorithms helps malware detectors to run faster and consume less system resources (Zhang *et al.*, 2009; Van Lunteren *et al.*, 2006; Amamra *et al.*, 2012a).

Table 2.2 Samples of SHA-1 signatures of smartphone malwares.

Malware name	**Package name**	**SHA-1 signature**
DroidDream	Com.droiddream.bowlingtime	72adcf43e5f945ca9f72 064b81dc0062007f0fbf
Geinimi	Com.sgg.spp	1317d996682f4ae4cce6 0d90c43fe3e674f60c22
Fakeplayer	Org.me.androidapplication1	1e993b0632d5bc6f0741 0ee31e41dd316435d997

Static signature-based detection techniques are very efficient and reliable to identify known malwares. However, they cannot detect unknown malwares and variants of known malwares. In addition, these techniques need human expertise intervention to develop signatures of new malwares, which is time consuming. Also, they are easily evaded by using different obfuscation techniques, such as binary packers, encryption, or self-modifying code, which means recycling existence malwares with different signatures (Christodorescu and Jha, 2006; Idika and Mathur, 2007; Rastogi *et al.*, 2013)

Static signature-based detection techniques are light in term of computational complexity. Therefore, they are suitable to smartphone systems. They adopt on-device architecture where the anti-malware is installed on the smartphone system. The main advantage of on-device architecture is that it is independent to external server, which means there would not be any server down problem and network congestion.

2.4.2 Behavior signature-based detection techniques

Behavior signatures are complex meta-structures embracing dynamic concepts and a semantic interpretation. Behavior signature is more effective and efficient to deal with obfuscation techniques, such as polymorphic, binary packers, and encryption. Analysis approach rule classifies behavior signature into static behavior signature and dynamic behavior signature. In static behavior signature, the signature is extracted by analysing the malware code. In dynamic behavior signature, the signature is extracted from runtime information by executing and monitoring the malware code (Amamra *et al.*, 2012b).

2.4.2.1 Static behavior signature-based detection techniques

These techniques are based on static code analysis technique, which uses information embedded in a given executable file or code templates to capture the functionality of a specific malware family. Blasing *et al.* (2010) have presented Android Application SandBox framework for malicious software detection. It performs static and dynamic analysis. The static analysis runs on the device, decompresses the apk file, converts their class files into java source code, searches for suspicious patterns and marks them as benign or malicious. Following a "trial-and-error approach", the authors defined the following malicious patterns: (i) usage of the Java Native Interface, (ii) usage of *Sytem.get* and *Runtime.exec()*, (iii) usage of reflection, (iv) usage of services and IPC provision, and (v) usage of Android Permissions.

The advantages of static behavior signature technique include: (1) it detects entire family of malware variants with one signature, (2) it detects malware before execution and all execution paths are enumeratively available. However, the technique suffer from limitations, such as (i) the features extraction is quite complex and requires several processing steps, (ii) it is computationally expensive because of disassembling and scanning large files, and then applies complicated classification algorithms. Therefore, static signature extraction requires much system resources.

2.4.2.2 Dynamic behavior signature-based detection techniques

These techniques are based on features extracted from runtime behavior. Since the malware execution shows the real malware intensions and activities, the dynamic signature behavior representation is closer to the real behavior of malware.

Among published researches Dai *et al.* (2010) proposed a technique that uses API interception techniques to dynamic analyse application's behavior. They analysed application's behavior compared to malware behavior signatures database. The malware behavior signature is a sequence of API function calls and is modelled by a Finite State Automaton (FSA). Push-Down Automata (PDA) has been used to describe the code samples to be analysed. The technique matches the malicious signature using model checking by computing the intersection of the FSA and PDA. This technique is tested on HTC PPC6800 running Windows Mobile 6.0 OS and compared with commercial Antivirus. While commercial anti-viruses can detect the virus before packing, they could not do it when the virus is packed. The proposed technique can detect those viruses before and after packing.

Bose *et al.* (2008) proposed a signature behavior detection approach. This approach is based on modelling application behaviors using temporal logic of causal knowledge (TLCK). They specify the behavior of malware as collection of resource-access and events made by that malware. The behavior monitor agent supervises the application behavior to construct on-line behavior signatures from APIs calls. They used SVM machine learning classifier to detect malwares from their partial or incomplete behavior signatures. The technique is tested on Symbian OS phone and it identifies current mobile viruses and worms with more than 96% accuracy.

Kim *et al.* (2008) proposed a signature-based power-aware malware-detection framework. It detects previously unknown energy depletion threats. The framework consists of a power monitor agent and a data analyser agent. The power monitor agent supervises the smartphone system and takes samples of power consumption, which are used to build a power consumption history. The data analyser receives the power consumption history from the monitor agent and its main job is to extract a unique pattern from the history to identify power signature. This power signature is then compared against the database of priori signatures. The data analyser uses two data-processing software components: noise-filtering and data-compressing. The framework was tested on an HP iPAQ running a Windows Mobile OS. It had high rate (up to 95%) storage-saving without losing the detection accuracy and high true positive rate (99%) in classifying malwares.

The advantages of dynamic signature-based technique are: (i) the signature is generic, where a single behavior signature detects not only one malware but a whole class of malwares, and (ii) dynamic signature is more close to malware behavior and intensions. The limitations of this technique are: (i) malicious behavior needs to be well analysed, (ii) dynamic behavior signature must be accurate and compact as much as possible, and (iii) the behavior signature cannot detect a new malware of a new behavior.

Dynamic signature-based technique generally adopts client-server architecture, where the applications are executed on smartphone system and their traces and events are sent to external server to extract the behavior signature.

2.5 Anomaly-based detection techniques

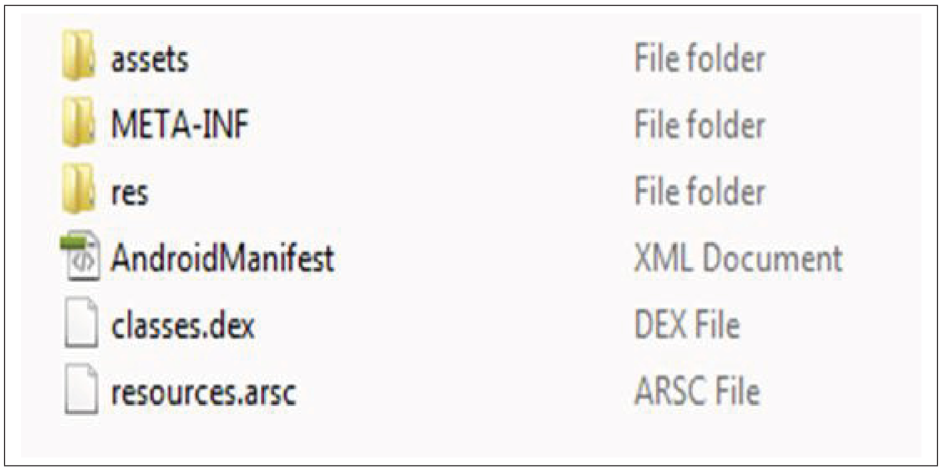
Anomaly-based detection techniques are broadly classified into static techniques and dynamic techniques. The static techniques use information embedded in a given executable file or code templates to capture the functionality of the application. The dynamic techniques use the events, traces, logs and information collected from the application's execution. This Thesis is mainly concerned with anomaly-based detection techniques on Android smartphone system.

2.5.1 Static anomaly-based detection techniques

The static techniques are performed without actually running the application. The analysis is based on static information that is extracted from the application package files. The mainly investigated static techniques in the prior work are: (i) modelling benign behavior profile using machine learning classifiers and static data, (ii) static dataflow analysis, (iii) control flow analysis and (iv) byte code inspection. These techniques are explained in more details in the coming of this subsection.

To understand the static techniques used for Android platform, we briefly review Android application package structure and the contained information. Android application is packaged in APK file. The APK file contains all the information necessary to install and run the application on the Android device or emulator (developer.android.com, 2014b). The components of an APK file are illustrated in Figure 2.8, and these are:

- *Assets* folder contains documents that inform about the application such as FAQ and license information.
- *META-INF* folder contains files such as CERT.RSA, the certificate of the application and CERT.SF to ensure the integrity of the APK package.
- *Res* folder holds resource files, such as videos and sounds.
- *AndroidManifast.xml* file presents important information about the application, such as the access permission, the API version and the references to library files. These information are essential to the Android system before it can run the application code.
- *Classes.dex* is Dalvik virtual machine executable byte code.
- *Resources.arcs* is binary resource file after compilation.



assets	File folder
META-INF	File folder
res	File folder
AndroidManifest	XML Document
classes.dex	DEX File
resources.arsc	ARSC File

Figure 2.8 The component of apk package

To extract features and information from an APK package, various reverse engineering steps are used, as shown in Figure 2.9. APK package can be decompressed by any archive tool, such as *winzip* into separate files and folders. The most used files in static analysis are the *Androidmanifest.xml* file and the *classe.dex* file. Androidmanifest.xml file is recovered into readable form using different reverse engineering tools, such as *apktool*. Various analyses are then applied to extract important information, such as requested permission. This information is then used to distinguish benign applications from malicious ones.

The *classe.dex* file is the byte code of the application. This file is disassembled into one of the two readable forms: into *smali* by using *apktool*, or into *java* by using *dex2jar* tool. After that, information mining and analysis are applied to extract features and data that can be used to detect malicious behavior.

In general, static techniques can be divided into two categories. First, train and test machine learning classifiers by static data, such as requested permissions and API. Second, using traditional code analysis. These techniques are reviewed in details in the next parts.

2.5.1.1 Machine learning classifier

Machine learning classifiers have been widely used in malware detection techniques. They are usually used to predict application's behaviors. Classifiers predict unknown instances as either benign or malicious. This process consists of two phases: training and testing. In the first phase (Figure 2.10 (a)), a training data of benign and/or malicious samples is provided to the classifier. By processing the training data, the classifier determines their parameters. In the testing phase (Figure 2.10 (b)), new benign and/or malicious samples that did not appear in the training samples set are provided to the classifier to evaluate its performance (Shabtai *et al.*, 2009b).

In prior work, different classifiers are implemented and tested and various data are used to represent behavior of apllications. The main investigated static data are: requested permissions, features and APIs.

Rapport-Gratuit.com

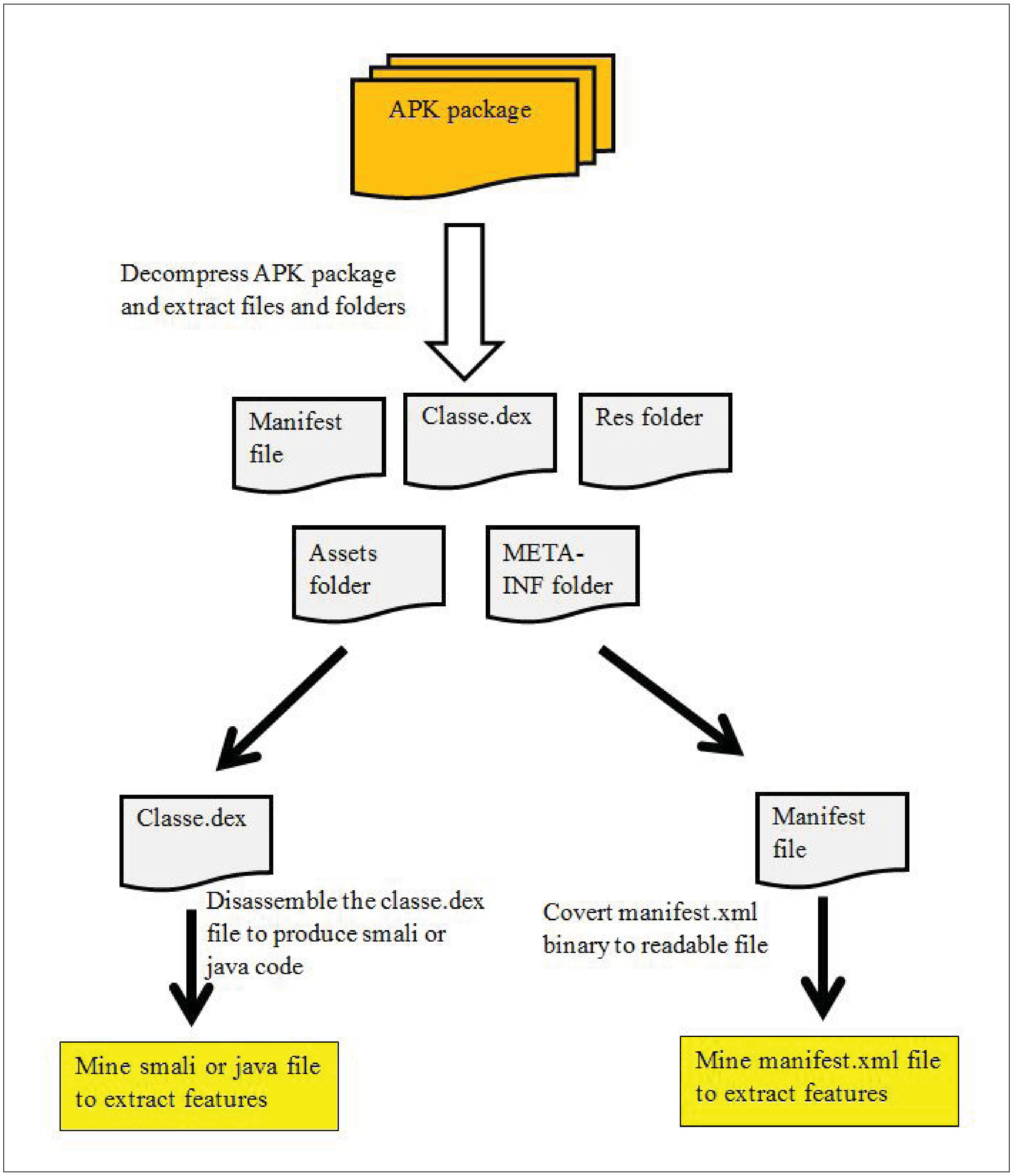

APK package
Decompress APK package and extract files and folders
Manifest file
Classe.dex
Res folder
Assets folder
META-INF folder
Classe.dex
Manifest file
Disassemble the classe.dex file to produce smali or java code
Covert manifest.xml binary to readable file
Mine smali or java file to extract features
Mine manifest.xml file to extract features

Figure 2.9 Overview of static techniques steps

- **The requested permission** : Android application cannot access most system resources, such as camera functions, network connections, telephony capabilities, SMS/MMS func-

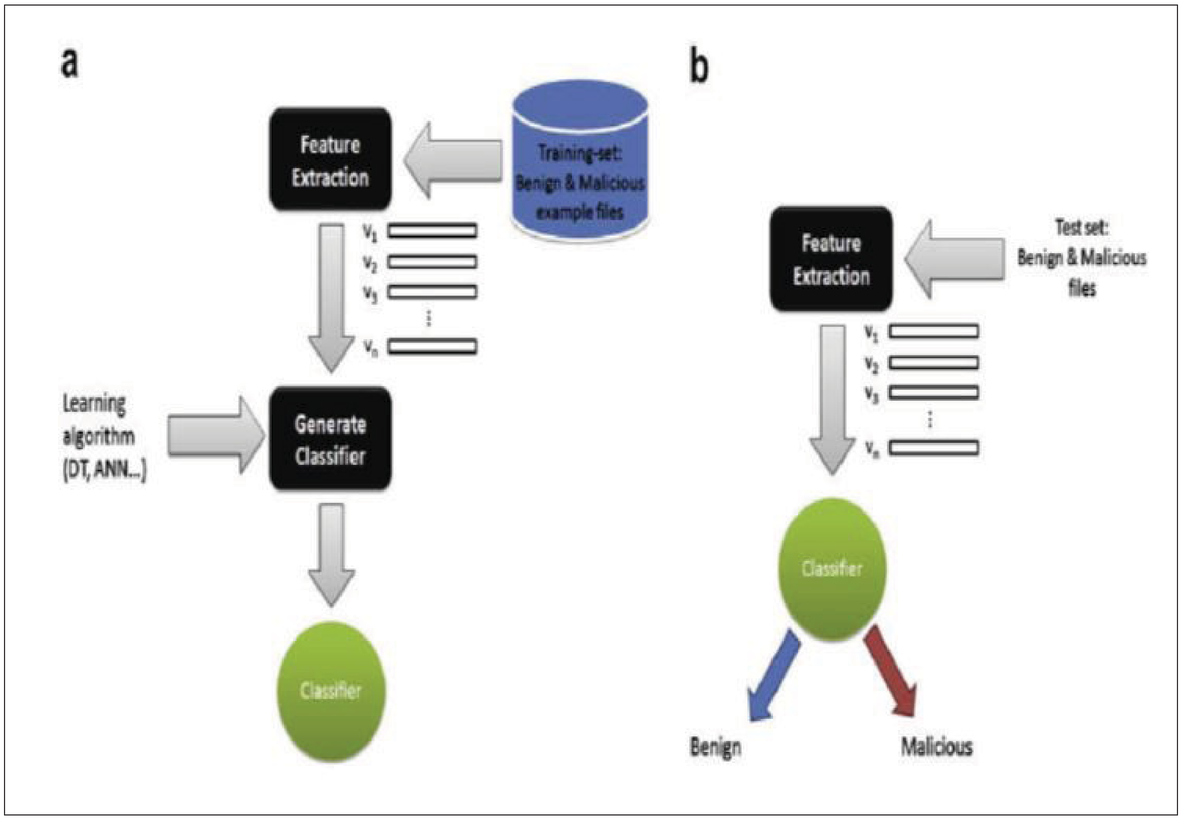

a
b
Feature Extraction
Training-set: Benign & Malicious example files
V1
V2
V3
Vn
Learning algorithm (DT, ANN...)
Generate Classifier
Classifier
Feature Extraction
Test set: Benign & Malicious files
V1
V2
V3
Vn
Classifier
Benign
Malicious

Figure 2.10 Machine learning classifiers steps
Taken from (Shabtai *et al.*, 2009b)

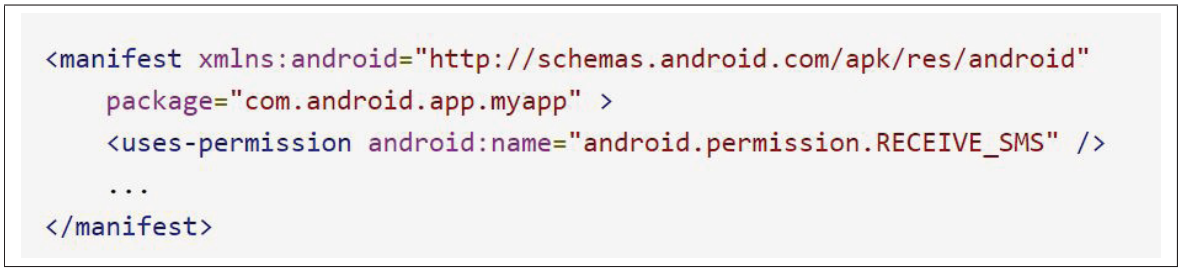
tions and reading or writing user's private information as well as other application files. In order to access these resources, Android application should declare the permissions it needs in the *AndroidManifest.xml* file using <*uses-permission*> tag (developer.android.com, 2014a) as illustrated in Figure 1.1. Android 2.3 has 134 permissions (Bartel *et al.*, 2014).

Among published papers Sahs and Khan (2012) extracted list of the requested permissions from the *AndroidManifest.xml* file, and they generated a binary vector where each entry corresponds to a built-in permission which is set to 1 if the application requests that permission, and to 0 otherwise. They train a one-class support vector machine (SVM) classifier only with benign applications because they have far more benign applications than malicious ones. The experiments have shown promising results.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

Aung and Zaw (2013) used the same data vector as (Sahs et Khan, 2012) with different machine learning classifiers. They applied k-means as first stage on training data, which contains benign and malware samples to obtain k disjoint clusters. In the second stage, they applied decision tree on the k clusters. The classifiers are evaluated by true positive rate, false positive rate and the overall accuracy. The experiment results show very good accuracy results.

Sanz *et al.* (2013) have examined the SVM, decision tree, k-nearest-neighbors, Bayesian network and random forest classifiers on the following dataset: requested permission, features and the combination (requested permissions + features). To measure the accuracy performance of the classifiers, they used four metrics: true positive rate, false positive rate, accuracy and area under the ROC curve (AUC). The experiment results good accuracy results for all classifiers. The classifiers are more accurate when using combination dataset. Random forest classifier has the better accuracy performance.



```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.android.app.myapp" >
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
    ...
</manifest>
```

Figure 2.11 Permission request in Androidmanifest.XML file

- **Features**: if an application requires to access a hardware or a software feature, it must declare that feature with a <*use-feature*> element in the *Androidmanifest.xml* file . Sanz *et al.* (2013) reported that the number of permissions required by benign and malicious applications are equal. Therefore, they conclude that the number of permissions does not seem to be accurately determining whether an application is a benign or not. They consider the *features tag* to add complementery information to permissions and provide closer behavioral view of the inspected application. They examined the combination of requested permission and features tags on five different classifiers: k-nearest neighbours, decision tree, Bayesian network, random forest and support vector machine. The results showed that combination gives better performance than using requested permission or feature tags alone.

- **API (Application Program Interface)** : it is a set of functions and tools for developing applications. Android provides rich framework API that allows developers to create powerful applications with innovative user interface and reliable access to real hardware. However, some sensitive functions are restricted by an intentional lack of API, such as APIs to manipulate SIM card. Other sensitive functionalities should be used by trusted applications and should be protected by permissions, such as camera functions, telephony functions and network functions (source.android.com, 2014). Some of these sensitive APIs are illustrated in Table 2.3.

Table 2.3 Sample of sensirive APIs.

API	**Function**
Camera.shutterCallbeck	Used to signal the moment of actual image capture
Camera.ErrorCallback	Camera error notification
Camera.PictureCallback	Used to supply image data from a photo capture
DisplayManager.DisplayListener	Listens for changes in available display devices
InputManager.InputDeviceListener	Listens for changes in input devices
LocationListener	Used to receive notification from Location Manager when the location has changed
GpsStatus.NmeaListener	Used for receiving NMEA sentences from the GPS

API calls are used by different researchers to detect malwares. Aafer *et al.* (2013) rely on API calls information within the bytecode. They statically analyzed a large number of malicious and benign applications and produced a list of distinct API calls within each category. A distinct API refers to a distinct combination of *Class Name, Method Name*, and *Descriptor*. They, then, conducted a frequency analysis to select those APIs which were more used in the malicious than in the benign set. They evaluated different standard classifiers: decision tree, k-nearest neighbours and support vector machine on API feature set and permission feature set. The classifier performance is measured by true positive ratio, true negative ratio and the overall accuracy. The experiment results show the APIs dataset has better performance than permissions dataset, and the KNN classifier has the best accuracy performance.

Peiravian and Zhu (2013) evaluated support vector machine, decision tree and bagging classifiers using the requested permissions, API calls and their combination. The dataset was extracted from 610 malicious applications and 1250 benign applications. Combing permissions and API calls had better classification performance because it provided additional information in characterizing the behavior of applications. The bagging classifier outperformed the other classifiers due to its capabilities to overcome imbalanced dataset.

Finally, Arp *et al.* (2014) presented DREBIN, a lightweight framework for Android malware detection. DREBIN performs static analysis on APK package to extract various static information. They focused on *Androidmanifest.xml* file and disassembled dex bytecode. The features extracted from *Androidmanifest.xml* file are: hardware components, requested permission, application component and filtered intents. The features extracted from disassembled bytecode were: restricted API calls, used permissions, suspicious API calls and network addresses. The dataset contained 123453 benign applications and 5560 malicious application. They used linear support vector machine classifier. DREBIN had a very good performance with high malware detection rate and low false positive rate.

2.5.1.2 Code analysis

Code analysis is the process of inspecting the application bytecode to understand the application's behavior. The code analysis mainly used in: checking for code errors, code improvement and optimization and security flaws. There are various static code analyses techniques used to detect anomalous behavior for Android applications. The main used ones are: control flow graph, data flow and bytecode instruction analysis. These techniques are explained in more details in the next paragraphs.

- **Control flow graph (CFG)**: CFG is an abstract representation of a program using graph notation. The nodes in the graph represent basic blocks, i.e., a sequence of consecutive instructions without any branching in or out in the middle of the instruction. The edges are used to represent the possible control flow from end of one block to the beginning of the other. Figure 2.12 illustrates simple CFG of simple code.

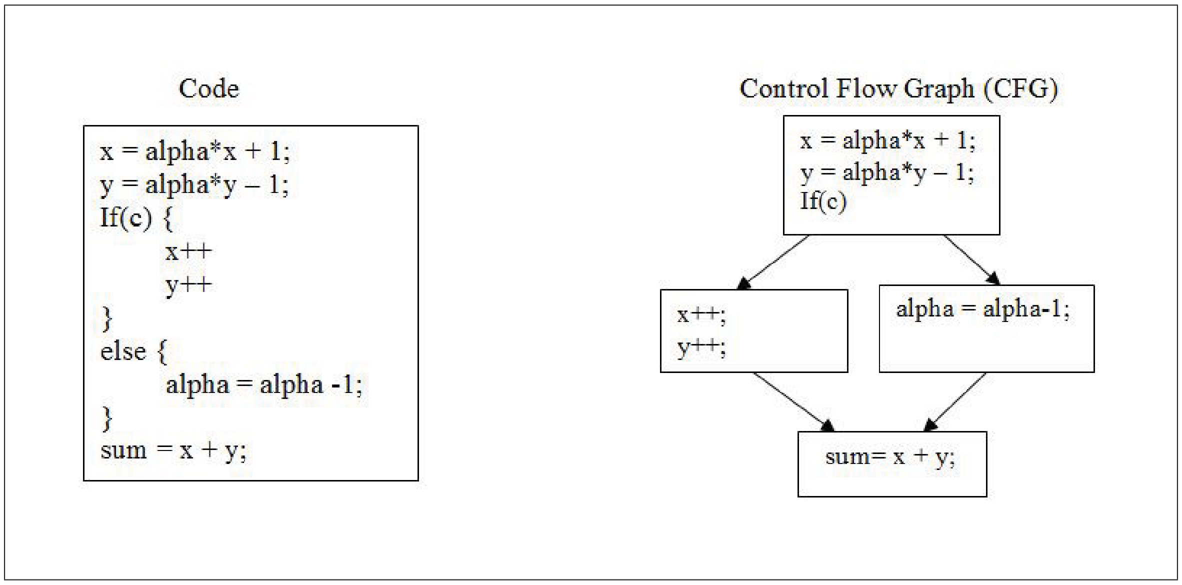


Figure 2.12 Control flow graph of simple code

CFG is mainly used in compiler optimization and static code analysis. However, its uses in the security bugs, vulnerability and anomaly detection keep increasing. Among the published researches Sahs and Khan (2012) extract CFG for each method from its raw bytecode. They, then, combine methods graphs into a large disconnected graph. Chin *et al.* (2011) developed ComDroid tool to detect potential vulnerabilities in Android applications. The tool uses control flow analysis to identify unauthorized Intent receipt attacks. Grace *et al.* (2012) propose Woodpecker, based on control flow to detect capability leaks in stock Android smartphone systems. They built CFG for each entry point. The whole application CFG was the union of all the entry points CFGs. They used the CFG and the entry points to locate possible execution paths and find the unprotected ones. Finally, Gibler *et al.* (2012) proposed a static analysis framework, AndroidLeaks, to identify any leaks of personal information by constructing the call graph of the application code.

- **Data flow**: it is the tracking of data flows through and across application components. Dataflow analysis are generally used to detect private information leakage. Among the published research, Aafer *et al.* (2013) use data flow analysis on specific code class on parallel with APIs frequency to increase the discrimination property of the proposed detection technique.

  Fuchs *et al.* (2009) developed a tool named, ScanDroid. Basically, this tool statically tracks data flows for Android application. ScanDroid extracts security specifications from *AndroiManifest.xml* file of the application, and verifies whether or not the data flows through the application code are consistent with those specifications.

  Arzt *et al.* (2014) proposed FlowDroid tool. FlowDroid parse different Android application files, including the dex class file, the layout xml files and the *Androidmanifest.xml* file defining the application component (activities, services, broadcast receivers and content) to retrieve lifecycle, callback methods and calls to sources and sinks in the application. After that, the main method is generated from the previous elements. This main method is used to construct call graph and inter-procedural control-flow graph (ICFG). The taint analysis tracks taints by traversing ICFG.

- **Bytecode instruction**: each Android application package consists of *classes.dex* file. This file contains the actual Dalvik bytecode for execution. Dalvik disassemblers such as *baksmali* are used to disassemble *classes.dex* file. Among the published research, Zhou *et al.* (2012) used bytecode information to identify repackaging applications. Repackaged applications are mainly used to insert malicious code. They developed DroidMOSS tool that measures the similarity between two applications. Each application is presented by a fingerprint. The fingerprint is generated by applying fuzzy hashing technique on application bytecode and developer's certificate.

Enck *et al.* (2011) statically analysed 1,100 popular free Android applications from Android Market (official Android applications Market). They implemented a Dalvik decompiler ded, which recovered Android application source code. Twenty-one million lines of code retrieved from 1,100 free applications were then analysed for exploring coding security failures, identifying misuse security information and uncovering vulnerabilities. They used four techniques to evaluate the code: control flow analysis, data flow analysis, structural analysis, and semantic analysis. The analysis results showed that besides an overwhelming concern for misuse of private and sensitive information such as phone number, IMEI and IMSI and ICC-ID, about 51% of the applications used unexpected number of *Ads* and analytics libraries. Also, it showed that many application developers did not follow necessary security precautions and guidelines. For example, sensitive data were often written to Android's centralized logs.

The main advantage of using static data is its ability to detect the malwares before the execution. However, the static features suffer from some limitations: the permissions declared in the Android manifest file do not necessarily mean that they are actually used within the code. Felt *et al.* (2011) reported that a large number of Android applications were over-privileged. Malicious behavior can be performed without any permission (Grace *et al.*, 2012). Frequent APIs is not an accurate discriminator between the malicious and the benign applications (Aafer *et al.*, 2013). Data flow, control flow and bytecode are quite complex and require several reverse engineering processing steps, and it is computationally expensive because of the disassembling and scanning big files.

2.5.2 Dynamic analysis

Dynamic techniques refer to the techniques that derive the behavior of an application while it is executed. These techniques can monitor different features of application execution, such as API calls, system calls and system resources. The application execution shows the real intensions and activities of the application. Therefore, the dynamic features should be closer to the real application behavior. The dynamic features are presented at different levels of application execution, as illustrated in Figure 2.13. At application level, dynamic features such as API calls can be intercepted. System calls represent features at kernel level. At hardware level, features about device hardware such as CPU memory can be extracted. For a reliable and efficient dynamic analysis, safe and reliable environment for running and analysing application is required. The two main used dynamic techniques are: machine learning classifiers and dynamic taint analysis.

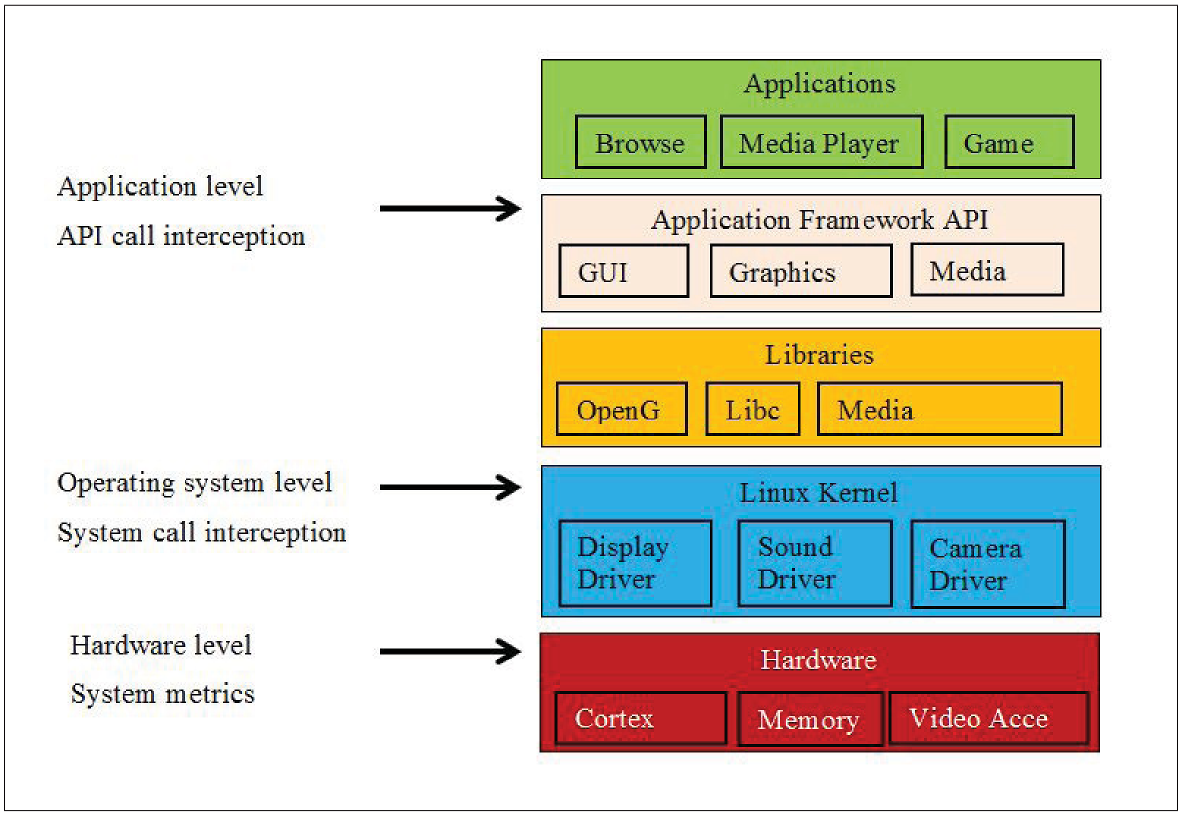


Figure 2.13 Basic levels of dynamic approaches

2.5.2.1 Machine learning classifier

Section 2.5.1.1 explained the main used static data to train and test machine learning classifiers for detecting anomaly. This section explains the main dynamic data that are used for the same purpose. Various data are investigated, such as system calls and system metrics.

- **System metrics**: these metrics measure system stability. The system has relatively stable performance and state when it is not under an attack. Once the system is infected by malware, the system shows abnormal behavior, such as slowness, excessive network communications and battery drain. Therefore, the system metrics are useful for malware detection. Broadly, Table 2.4 presents the most used metrics. The system continuously monitors its performance and extracts the representative parameters; it then applies machine learning classifiers to classify them as benign or malicious. Features extraction complexity varies from one feature to another. For example *RAM_free* feature can be easily extracted using *ActivityManager.getMemoryInfo()* API. However, there are no APIs to read *Process_count* and *CPU_usage* features. Therefore, the extraction process of these features is more complex and needs algorithm to calculate the values.

Table 2.4 Samples of system metrics.

Feature	**Description**
CPU_usage	The usage of CPU
RAM_Free	The available memory
Battery_consumption	The consumption level of battery
Process_count	The number of running processes
Install_app	The number of installed apps
SMS_out	The number of SMS sent
MMS_out	The number of MMS sent
Call_in	The number of incoming calls
Call_out	The number of outgoing calls
WiFi_traffic	The WiFi network traffic

Among the published research based on system metrics, Schmidt *et al.* (2009) proposed a framework to monitor smartphone system running Symbian OS and Windows mobile OS

Rapport-Gratuit.com

in order to extract system features used in the process of detecting anomaly applications. The proposed framework is based on monitoring client, which runs on the smartphone system, collects the data describing the system state, such as the amount of free RAM, the number of running processes, CPU usage, and the number of SMS messages in the sent directory, and sends them to Remote Anomaly Detection System (RADS). The remote server contains a database to store the received features. The detection unit accesses the database and runs machine learning algorithms, e.g. AIS or SOM to distinguish between benign and abnormal behaviors. A meta detection unit weighs the detection results of different algorithms. The algorithms were executed on four features sets of different sizes, reducing the set of features from 70 to 14 features, and thus, saving 80% of disk space and significantly reducing computation and communication cost. Consequently, the approach had a positive influence on the battery life and a small impact on true positive detection.

Shabtai *et al.* (2012) used various system metrics, such as CPU consumption, number of sent packets through WiFi, number of running process and battery level to evaluate and compare the performance of the following machine learning classifiers: k-means, logistics regression, histograms, decision tree, bayesian networks and naïve bayes. Author performed four experiments using 40 benign applications (20 games and 20 tools), and 4 proof-of-concept malicious applications with different objectives. Experiment 1 and Experiment 3 evaluated the ability of classifiers to differentiate between malicious and benign applications when the testing dataset is included in the training dataset. However, in experiment 1, the training and testing processes were performed on the same device, while in experiment 3 they were performed in two different devices. Experiment 2 and experiment 4 evaluated the ability of classifiers to differentiate between malicious and benign applications when the testing dataset was not included in the training dataset. In experiment 1, the training and testing processes were performed on the same device, while in experiment 4, they were performed in two different devices.

Yuan *et al.* (2013) proposed a malwares detection framework for Android system. The framework consists of these components: data extraction module, data analysis engine and response module. The data extraction module monitors the system activities and extracts the following features: CPU usage, battery consumption, memory usage, the amount of running processes, the amount of running thread, installed application, the inflow of network traffic, the outflow of network traffic, the amount of sent SMS and the amount of sent MMS. Data analysis engine analyzes the extracted data by naïve bayes classifier for determining if there is any anomalous behavior. Based on the analysis results, the response module takes the appropriate action on Android smartphone system. The used dataset consists of 45 benign applications and 15 malicious applications. The results of experiments showed good detection rate.

- **Power consumption**: the relationship between power consumption and malware detection comes from the idea that malicious applications use system resources more than benign applications, especially communication channels, such as Bluetooth, WiFi and 3G technologies. Therefore, system under attack and benign system should have different power behavior.

  Among the published papers, Buennemeyer *et al.* (2008) proposed B-SIPS (Battery-Sensing Intrusion detection Protection System) that uses battery constraints to detect malware activities. B-SIPS monitors the smartphone system power consumption with two communication channels Bluetooth and WiFi. The anomalous activity is detected if the power consumption level exceeds the system's dynamic threshold value. B-SIPS alerts the user and sends a report to the CIDE server. The Correlation Intrusion Detection Engine (CIDE) server receives data from the smartphone system and creates a profile to track the battery power and the number of running processes while the smartphone system is not under attack. The average and standard deviation are computed for each system and an alert to SA is generated if the system reports a value that is higher than the average and standard deviation. B-SIPS has low positive rate because of Dynamic Threshold Calculation (DTC) algorithm used to calculate the power consumption value dynamically.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

- **System calls**: Linux system calls are the interface between user applications and the kernel services, as illustrated in Figure 2.14. System calls provide functions to user applications, such as file operations (open, read, write, etc.), network operations (connect, send, receive, etc), or process operations (create a new process, kill process, etc). System calls tarce provide useful information about application behavior. Therefore, it is used in the field of anomaly detection.

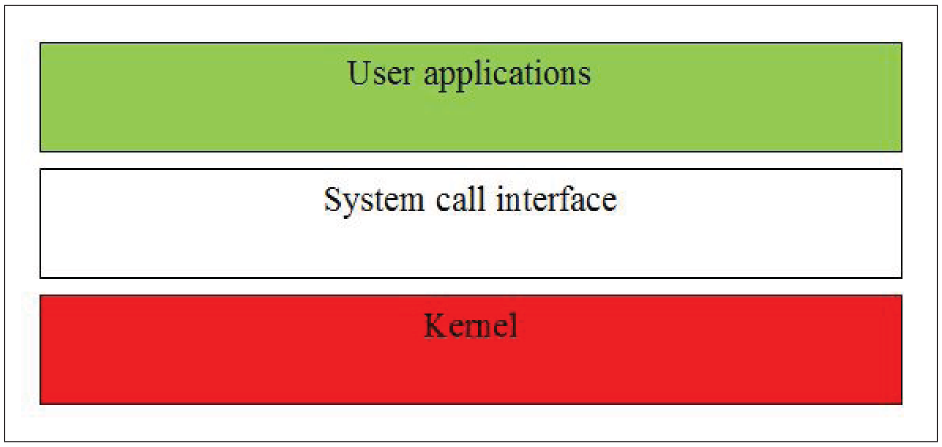


Figure 2.14 System call interface block diagram

Among the published papers, Burguera *et al.* (2011) proposed a solution based on Linux system calls. This solution contains several components. Crowdroid is a lightweight program that is executed on the system to collect system calls and send them to a remote server. In the remote server, the system calls data is analyzed and feature vector is created for every application. To differentiate between malicious and benign applications k-means algorithm is applied. The solution is tested by 50 benign applications and 10 malicious applications and show good performance.

Lin *et al.* (2013) propose SCSdroid framework to detect Android malware using sequence of system calls. SCSdroid uses thread-gained system call sequences instead of process-gained system call sequences. Thread-gained system calls means that the system calls produced by the process and its threads are recorded separately. SCSdroid adopts Bayes Theorem to distinguish between malware and benign applications. It is evaluated by a dataset that consists of 25 malicious applications and 100 benign applications, and it shows good accuracy performance.

2.5.2.2 Dynamic taint analysis

Taint analysis techniques (Lokhande and Dhavale, 2014; Newsome and Song, 2005) refer to tracking the information flow through an application and possible leakage. Taint analysis is widely used in the security applications, such as unknown vulnerability detection, malware analysis and track flow of sensitive information. In the taint analysis, the sources and sinks of sensitive information are predefined. Taint sources are the sources of sensitive information, such as accounts, emails, address book, calendar, personal database or files, phone state, SMS/MMS messages and Unique identifiers (IMEI). Whereas taint sinks are the destinations from where data can leak out of the system, such as internet transmission, SMS/MMS transmission, publicly accessible storage and inter-process communication message. Taint analysis identifies whether there are routes from sources to sinks. If sensitive data can reach a sink, it is identified as instances of data leakage. An overview of dynamic taint analysis is illustrated in Figure 2.15.

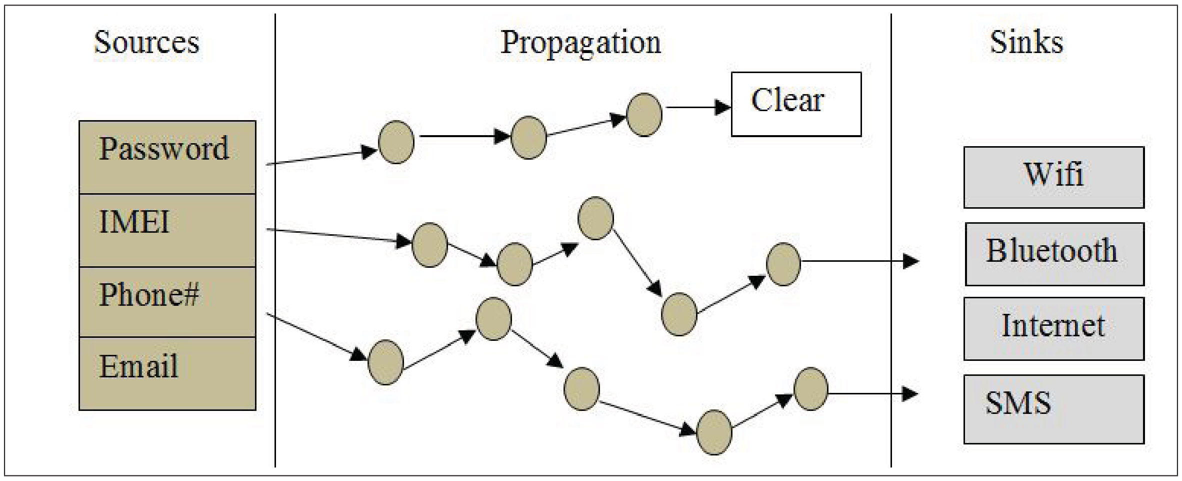


Figure 2.15 Overview of dynamic taint analysis

Among the published papers, Enck *et al.* (2014) proposed TaintDroid, a real-time system tracks privacy-sensitive data flow. TaintDroid monitors how a third-party applications access and manipulate private information. The user is warned when monitored information leaves the device. TaintDroid has the ability to track multiple sources and sinks simultaneously. It automatically taints data from sources and transitively applies labels as sensitive data propagates through program variables, files, and inter-process messages. When monitored data are leaked over the internet, SMS messages, Bluetooth or any other means, TaintDroid logs the data's labels, the application responsible for transmitting the data, and the data's destination. Such log helps security administrator to understand how the application handle sensitive data, and can potentially identify the applications of misusing these information.

TaintDroid evaluated on 30 popular Android applications that use location, camera, or microphone data. TaintDroid discovered that two-thirds of the examined applications used sensitive data suspiciously. Fifteen of the 30 applications reported users' locations to remote advertising servers and 7 applications gather the phone sensitive information: the device ID, the phone number and the SIM card serial number.

2.6 Conclusion

In this chapter we have reviewed smartphone malware detection techniques. We have attempted to compile and analyse common features and functionalities of detection techniques, and infer classification criteria. According to these criteria, we have provided structured and comprehensive malware detection techniques taxonomy. Techniques used for detecting malwares for smartphone systems are classified broadly into: signature-based detection techniques and anomaly-based detection techniques. This classification relies on reference behavior used by detection technique. Inside these categories, ramifications are then deduced according to analysis approaches, algorithms, and the used dataset.

Dynamic analysis is simpler than static analysis. It requires only isolated environment to install and execute the applications. *Android SDK tool* is a tool to develop Android applications and provides isolated Android system virtual machine, which can be used in dynamic analysis.

System call trace is simple to obtain and it does not need any application knowledge or instrumentation. System call trace is obtained by running simple Linux command*strace*. System call trace describes the application interactions with system. Therefore, it is representative of application behavior.

The most proposed techniques inherited from the known malware detection techniques used in computers and network where there are enough resources (enough memory space and powerful processors). The main challenge facing this research area is innovative techniques respecting smartphone resources and assures high detection accuracy and low false positive rate.

CHAPTER 3

SYSTEM CALLS AND FILTERING & ABSTRACTION PROCESS

3.1 Linux system calls

Linux kernel is the lowest layer of the Android architecture. Linux system calls are the interface between user applications and the kernel services. System calls provide functions to user applications, such as file operations (open, read, write, etc.), network operations (connect, send, receive, etc.), or process operations (create a new process, kill process, etc.). As illustrated in Figure 3.1, when a user application requires a service from the operating system, the request goes through the *glibc* library, the system call interface, the kernel and, finally, to the hardware. The *glibc* library interprets the requests and the CPU switches from user mode to kernel mode and executes the appropriate kernel function looking into the system call table. Afterwards the user application gets the information requested in the user space in an inverse process. Functions like getpid(), open(), read() and socket() are some of the functions that *glibc* can provide applications to invoke a system call (Burguera *et al.*, 2011).

Intercepting and analyzing the system calls provide useful information about the behavior of an application. Several researchers (Burguera *et al.*, 2011; Amamra *et al.*, 2012c, 2013) used system calls for anomaly detection for various reasons such as:

- System call traces collected from the executions give the basic operations of an application like creating or ending a process, opening or closing a file and making a network connection. This makes system call traces closer to the real intentions of the application.
- The analysis of system call traces looks at the application as a *black box*. It does not need any specialized knowledge of the application. System call traces can be obtained without the instrumentation of the binary code or its recompilation.
- System call traces are simple and represent good discriminators for several types of malware.

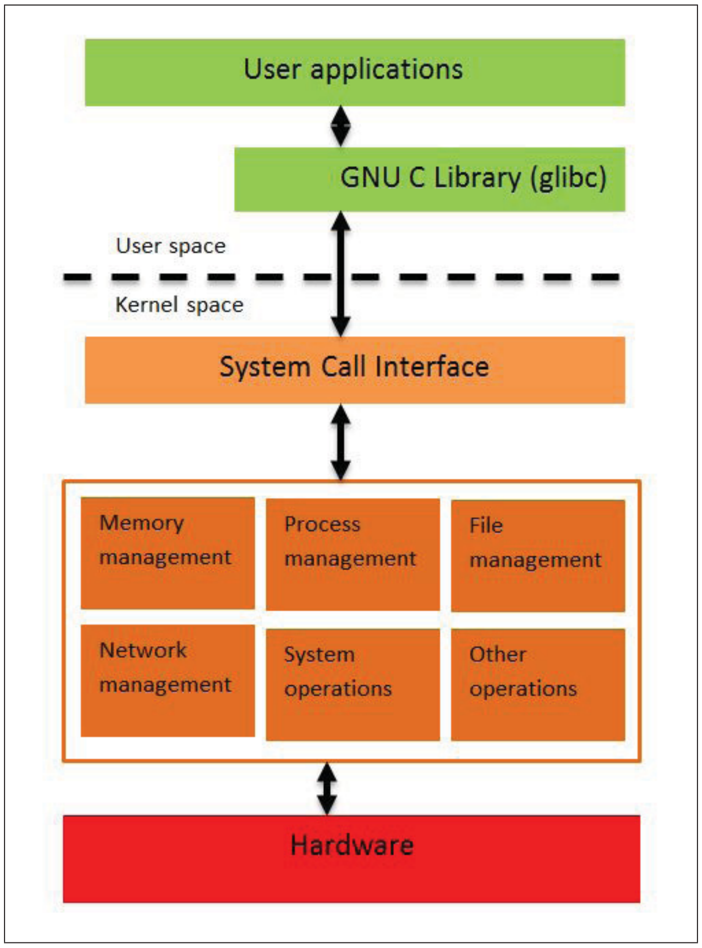


Figure 3.1 Linux system call execution

For the aforementioned reasons, our proposed solution is based on system call traces. However, the system call traces need more investigation and improvement to overcome the limitations mentioned in section 3.2.

This chapter presents our contribution on dataset level and it is twofold:

- The filtering and abstraction process is introduced to refine and simplify system call traces. This process is explained in details in section 3.3 and its performance is evaluated in Chapter 4 and Chapter 5.

- The feature vector representation is investigated to compact the dataset and enhance the accuracy performance of the classifier. This process is explained in details in section 3.4 and it is evaluated in section 3.4.4.

3.2 Limitation of using system calls

Anomaly-based detection systems based on system call traces still suffer from important limitations affecting their performance and adaptability. These limitations are summarized in the following subsections.

3.2.1 Easy evasion

Anomaly-based detection techniques based on system call traces can be evaded easily by inserting no-ops or nullified system calls, or replacing system calls with other system calls having the same functionality but different names (Wagner and Soto, 2002).

- *Inserting no-ops system calls*: malicious sequences can be altered by inserting no-ops system calls. For example, Figure 3.2 (a) illustrates an harmful sequence of four system calls and any malware detection system should identify this sequence. This sequence can be easily altered by inserting system calls having no effect like system calls returning information about process or file status, or returning time or clock status as presented in Figure 3.2 (b). Similarly, system calls doomed to failure can be used (e.g., opening non-existing files as presented in Figure 3.2 (c)).
- *Replacing system call with aliases*: malicious sequences can be altered by replacing system calls by system calls having the same functionality but different names. For example, pread(int fd, void *buf, size_t count, off_t offset) can easily replace read(int fd, void buf, size_t count) except that pread reads the file with a given offset.

Rapport-Gratuit.com

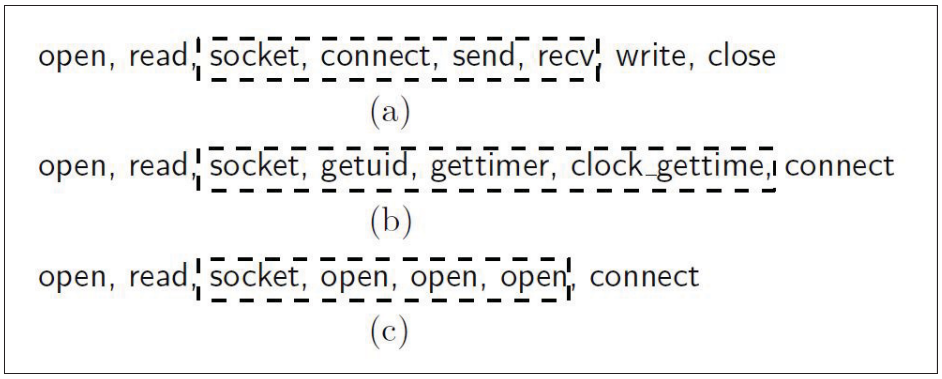

open, read, socket, connect, send, recv, write, close
(a)
open, read, socket, getuid, gettimer, clock_gettime, connect
(b)
open, read, socket, open, open, open, connect
(c)

Figure 3.2 Evasion techniques using no-ops system calls to hide malicious sequences

3.2.2 Complex application behavior

System call traces contain several system calls irrelevant to describe the main behavior of an application, such as system calls saving and maintaining process information, system calls checking resources availability and managing resources, inter-process communication, and memory management. These irrelevant system calls produce large and complex traces which may lead to inaccurate models of the application behavior.

3.2.3 Time and space complexities

To build accurate benign behavior models, anomaly-based detection techniques are usually based on machine learning and statistical approaches, such as support vector machine (SVM), hidden markov model (HMM), Bayes model and *k*-means. However, these approaches have two main limitations. First, their space and computational costs are generally expensive on large size of system call traces. In some cases, the algorithms require multiple passes over the entire datasets, thus violating the real-time aspect of an anomaly detection system. Second, most of statistical approaches assume the benign behavior does not change while the system is being trained and tested. This latter limitation violates the adaptability principle (Forrest *et al.*, 2008). Finally, the number of system calls keeps increasing from one OS kernel version to another and this impact the adaptability and complexity of the approach.

To overcome these limitations we propose a new process which filters and uses system call abstraction to simplify and reduce the size of system call traces. Moreover, the refined traces should be closer to the applications main behavior.

3.3 System calls filtering and abstraction process

The system call traces generated by applications correspond to low-level events. These traces represent the interactions between the application and the kernel, such as information maintenance, memory management, inter-processes communications, and hardware interrupts, etc.

Filtering and abstraction is the process of refining system call traces according to well defined rules and producing compact and more discriminative system call traces. Once the size of the traces has been reduced, the anomaly detection techniques would consume lesser computational resources and improve their detection accuracy (Amamra *et al.*, 2014).

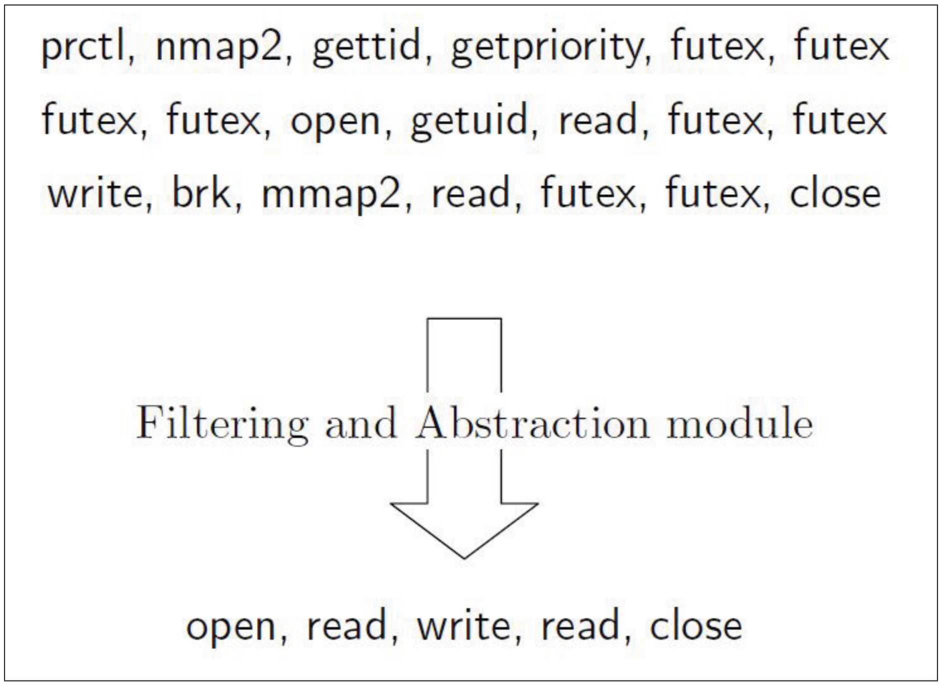

prctl, nmap2, gettid, getpriority, futex, futex
futex, futex, open, getuid, read, futex, futex
write, brk, mmap2, read, futex, futex, close
Filtering and Abstraction module
open, read, write, read, close

Figure 3.3 Overview of filtering and abstraction process

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

Figure 3.3 presents the filtering and abstraction process principle where the input is a raw system call trace and the output is the corresponding refined trace. The raw trace consists of all the system calls generated by the execution of the application, such as memory management (mmap2), get information system calls (gettid), etc, while the refined trace consists of only the system calls describe the application main behavior. This process consists of two phases: filtering phase that ignores all system calls irrelevant to describe the main application behavior and abstraction phase that unifies system calls having similar and/or overlapping functions.

3.3.1 Filtering phase

Filtering is the process of removing the system calls that are useless to characterize the main behavior of applications. For this purpose, we have studied Linux system call functions and categories. Linux system calls can be roughly grouped into six categories according to their functionality as in the succeeding text:

- File management: create and delete files, open and close files, read and write files, and get and/or set files status and attributes.
- Process management: create and terminate process, load or execute process, get or set process attributes, and wait event or signal.
- Memory management: allocate or free memory, and share or map memory and/or virtual memory operations.
- Inter-process communication(IPC): exchange data among processes.
- Network communication: make or delete communication connections, send and receive messages, get status information, and attach or detach remote devices.
- System wide: privileged and unprivileged file system, module management, and privileged miscellaneous.

From the aforementioned categories, we infer which system calls are irrelevant to describe the main behavior of Android applications. These system calls are grouped in the following four classes: inter-process communication, memory management, information maintenance system calls and unsuccessful system calls.

3.3.1.1 Inter-Process Communication

For security and safety purposes, each Android application runs in an isolated virtual machine. Thus, an application cannot manipulate directly the data of another application. However, in many cases, the data exchanges between applications are necessary. Android is based on an enhanced version of Linux containing a kernel module called *Binder* for inter-process communications. All *Binder* transactions happen through the ioctl system call (Hsieh *et al.*, 2013; Guojun *et al.*, 2012; Schreiber *et al.*, 2011). Thus, all the standard Linux inter-process communication (IPC) mechanisms can be ignored: pipes, message queues, semaphores, and shared memory. Table 3.1 summarizes the standard Linux inter-process communication mechanisms.

Table 3.1 IPC system calls.

IPC mechanism	**System calls**	**Nb. of system calls**
Pipe	pipe, pipe2	2
Message Queue	msgctl, msgget, msgsnd, msgrcv, mq_getsetattr, mq_notify, mq_open, mq_timereceive, mq_timesend, mq_-unlink	10
Shared Memory	shmctl, shmat, shmdt, shmget	4
Semaphore	semctl, semget, semopl, semtimedop	4

3.3.1.2 Memory management

The *malloc* C-function is usually implemented by brk and/or mmap system calls. The brk system call grabs a large chunk of memory and then split it as needed to get smaller chunks for the *malloc* C-function. The brk system call can change the size of the heap. The mmap system call is used when very large memory space is allocated. Calling mmap reduces the negative effects of memory fragmentation. As such, not all the implementations of the *malloc* C-function execute the same system call sequences. Two applications with the same behavior may have very different memory management system call sequences (Wang *et al.*, 2009; Waseem, 2010). Thereby, we can ignore these calls because they do not characterise the application main behavior. Table 3.2 presents the memory management system calls.

Table 3.2 Memory management system calls.

Memory mechanism	**System calls**	**Nb. of system calls**
Memory Management	mmap, mmap2, munmap, mremap, mlock, munlock, mlockall, munlochkall, set_mempolicy, brk, sbrk, mprotect, modify_ldt	13

3.3.1.3 Information maintenance system calls

Many system calls do not add any valuable information to describe the application behaviors. They exist simply for the purpose of transferring information between an application and the operating system. For example, system calls returning the current time and date (e.g., time, gettimeofday), system calls returning information about the system such as available or free system resources (e.g., getcpu, getrlimit), system calls returning file status (e.g., state, fstatfs, getxattr, listxattr). Also, system calls returning process information such as the user id and group id of the process (e.g., getuid, getgid), and system calls synchronizing the process (e.g., wait, epoll_wait) (Silberschatz *et al.*, 2012). Therefore, these system calls can be ignored. Table 3.3 illustrates the different classes of information maintenance system calls.

3.3.1.4 Unsuccessful system calls

Unsuccessful system calls do not have any impact on the application behaviors. For example, if an application fails to open a file in the first two attempts and succeeds in the third one,

Table 3.3 Information maintenance system calls.

Class	**System calls examples**	**Nb. of system calls**
File Management	mmap, mmap2, munmap, mremap, mlock, munlock, mlock-all, munlochkall, set_mempolicy, brk, sbrk, mprotect, modify_ldt,...	39
Process Management	getpriority, wait, poll, select, epoll_-wait, getuid, getgid, getgroups,...	55
Communication	getsockopt, getsockname, gethost-name, gethostid, getpeername, ...	6
Miscellaneous	futex, clock_gettime, clock_-nanosleep, getcpu, getpagesize, olduname, gettimeofday, ...	36

the open system call is used three times. In such a case, the first two open calls should be ignored (Wang *et al.*, 2009).

3.3.2 Abstraction phase

There are many system calls having the similar and overlapping functionalities but different names. For example, the readv(int fd, const struct iovec, int iovcnt) system call has almost the same function as read(int fd, void buf, size_t count). The only difference lies on the fact that the former call fills multiple buffers instead of only one. An example of overlapping functionalities, the open system calls can be used to open new files or existing files while the create system call can be used only to open new files. Hence, the abstraction process considers system calls of the similar and overlapping functionalities as equivalent. This process does not only simplify the traces and reduce the required resources to analyse them, but also protect against attacks where an attacker replaces system call with equivalent ones (Wang *et al.*, 2009). Table 3.4 illustrates some system calls abstraction.

Table 3.4 System calls aliases examples.

System call	Alias
open	open, openat, create, dup
read	read, readv, pread, readlink, readlinkat
write	write, writev, pwrite
recv	recv, recvfrom, recvmsg
send	send, sendto, sendmsg, sendmmsg
rename	rename, renameat
exec	exec, execve

3.3.3 End result

We examine Android OS version 2.3 based on Linux kernel version 2.6. This version has 317 different system calls (Inoue and Somayaji, 2007). The filtering process reduces the number of system calls by 169 and the abstraction process reduces the number by another 23. Hence, the refined traces would be composed of at most 125 different system calls. Table 3.5 illustrates some examples of application traces and their potential reduction after the filtering and abstraction process.

Table 3.5 Impact of the filtering and abstraction process on the size of application traces.

Application name	Nb. of system calls of raw trace	Nb. of system calls of refined trace	Improvement
Play music	18008	3373	81,26%
Firefox	7324	1360	91,43%
Calculator	42143	13477	68%
Meteo media	170177	41929	75.36%
Solitaire game	41244	16548	70%
Sodoku game	91783	8777	90.40%

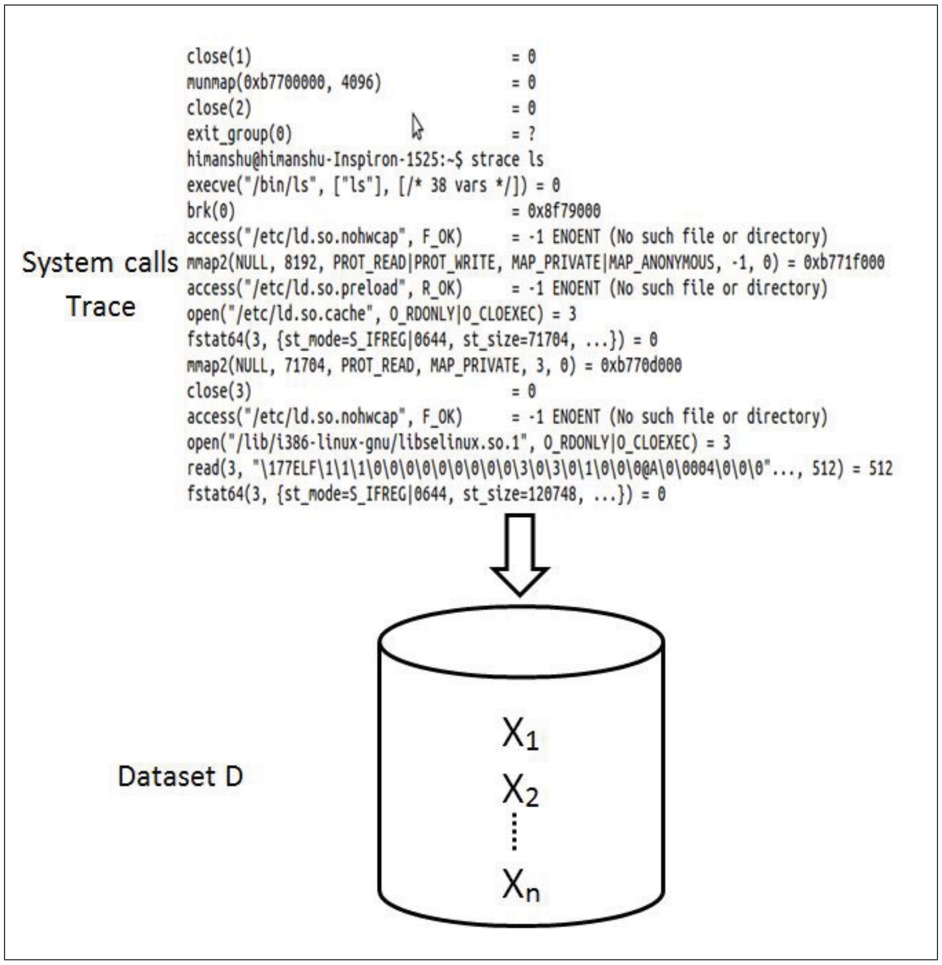
3.4 System call feature vectors

To improve the performance of anomaly-based detection techniques, prior works explored actively two factors: dataset (size and type) such as, requested permission (Aung and Zaw, 2013), API (Aafer *et al.*, 2013) and system calls (Amamra *et al.*, 2013); and the algorithm used to model the benign behavior profile such as, k-means (Burguera *et al.*, 2011), SVM (Bose *et al.*, 2008) and hybrid classifier (Amamra *et al.*, 2012c). Increasing the quantity of dataset or varying the type of used dataset to characterize benign behavior improves the accuracy of the benign behavior profile. However, the resulting profile has usually high computational and space overhead. Therefore, these factors are not suitable to improve performance of anomaly-based detection techniques on limited resources environment, such as smartphone systems. Thereby, exploring other factors that have significant influence on the performance of anomaly-based techniques as well as low computational and space complexities is an important step for anomaly detection techniques in smartphone systems. In this section, we highlight the dataset feature vector representation as a new factor.

Roughly speaking, a system call trace is a chunk of information. That information is formatted and organized in a dataset D as shown in Figure 3.4. The dataset D has a specific format that expresses how the data is organized. This format is called dataset representation or feature vector, and it is used during training as well as detection phases of machine learning classifiers. The process of preparing dataset feature vector representation has light computational and space complexity which makes it a suitable approach toward smartphone malwares detection technique.

In general, there are three feature vector representations. These representations are studied and compared in (Amamra *et al.*, 2013). Briefly, these representations are:

- Successive system calls representation, where the ordering information between system calls is considered in sequence. This dataset representation is used in prior work.
- Bag of system calls representation, where the successiveness of system calls in sequence is disregarded and only the frequency of each system call is preserved. It is used in previous works.



```
close(1)                              = 0
munmap(0xb7700000, 4096)              = 0
close(2)                              = 0
exit_group(0)                         = ?
himanshu@himanshu-Inspiron-1525:~$ strace ls
execve("/bin/ls", ["ls"], [/* 38 vars */]) = 0
brk(0)                                = 0x8f79000
access("/etc/ld.so.nohwcap", F_OK)    = -1 ENOENT (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb771f000
access("/etc/ld.so.preload", R_OK)    = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=71704, ...}) = 0
mmap2(NULL, 71704, PROT_READ, MAP_PRIVATE, 3, 0) = 0xb770d000
close(3)                              = 0
access("/etc/ld.so.nohwcap", F_OK)    = -1 ENOENT (No such file or directory)
open("/lib/i386-linux-gnu/libselinux.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0@A\0\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0644, st_size=120748, ...}) = 0
```

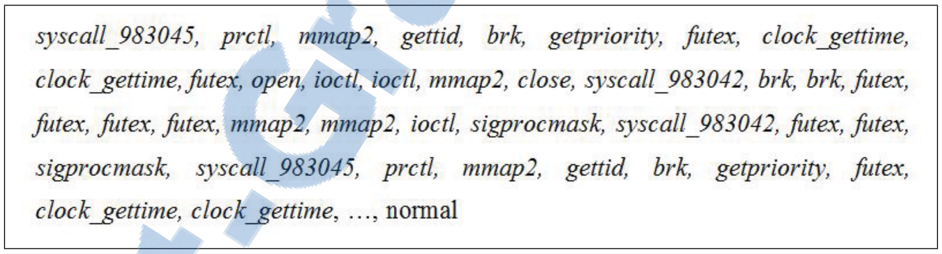
Figure 3.4 System calls trace transformation to Dataset

- Patterns frequency system calls representation combines features of the two previous representations. Pattern-frequency representation regards the successive order information of system calls in short pattern, and regards the frequency of each pattern in the sequence. To the best of our knowledge, this representation has not been studied in the past in this context.

The following notations are used hence forward. Consider the set of system calls $\sum = (s_1, s_2, s_3 \ldots s_m)$, where m is the number of system calls of operating system. Let T_i be a finite sequence of system calls and $|T_i|$ represent the length of the sequence. Let $\sum^*$ be the set of all possible finite sequences of system calls, $T_i \in \sum^*$. N is the number of applications (malware and benign) used in training and test phases. Let D be the set of of feature vectors $\{X_1, X_2, \ldots, X_N\}$ associated to the application traces $\{T_1, T_2, \ldots, T_N\}$. The feature vector X_i is defined according to the used representation as explained in the coming subsections.

3.4.1 Successive system calls feature vector representation

This feature vector representation considers the sequential order information of system calls in trace. Formally, this feature vector representation can be defined as follow: $X_i = \{s_{i,1}, s_{i,2}, \ldots, s_{i,|T_i|}\}$, Where $s_{i,j}$ is a system calls of order j in the trace T_i. Figure 3.5 shows an example of successive system calls representation.



syscall_983045, prctl, mmap2, gettid, brk, getpriority, futex, clock_gettime, clock_gettime, futex, open, ioctl, ioctl, mmap2, close, syscall_983042, brk, brk, futex, futex, futex, futex, mmap2, mmap2, ioctl, sigprocmask, syscall_983042, futex, futex, sigprocmask, syscall_983045, prctl, mmap2, gettid, brk, getpriority, futex, clock_gettime, clock_gettime, …, normal

Figure 3.5 Successive system calls feature vector

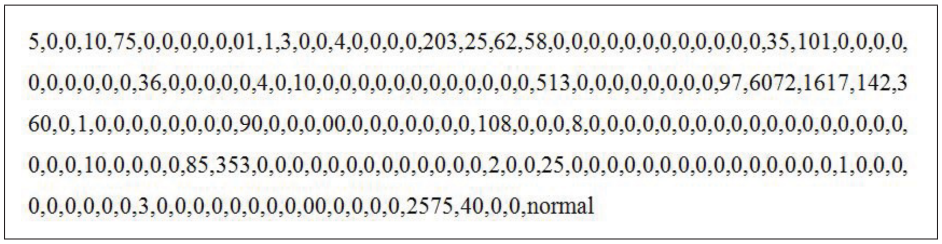
The space complexity of successive system call feature vector representation depends on the number of traces (N) and the total length of these tracs. The space complexity is given by $O(|T|)$. Generally, the total length of traces is large. Therefore, this representation is space consuming.

3.4.2 Bag of system calls feature vector representation

This feature vector representation disregards the ordering information of sequential system calls. Only the frequency of system calls in the sequence is maintained. Formally, the features vector can be defined as follow: $X_i = \{n_{i,1}, n_{i,2}, \ldots, n_{i,|\Sigma|}\}$, where $n_{i,j}$ is the number of occurrences of a system call s_j in the sequence T_i.

Rapport-Gratuit.com

Figure 3.6 illustrates bag system calls representation of a benign application. Each number in the sequence represents the frequency of a system call in the application trace. For example, the numbers 5,0,0,10,75. . . correspond to frequency of the following system calls: fstat64, setgroups32, setgid32, setuid32, getuid32 respectively, and the last attribute indicates the sequence of a benign application.



```
5,0,0,10,75,0,0,0,0,0,01,1,3,0,0,4,0,0,0,0,203,25,62,58,0,0,0,0,0,0,0,0,0,0,0,35,101,0,0,0,0,
0,0,0,0,0,0,36,0,0,0,0,0,4,0,10,0,0,0,0,0,0,0,0,0,0,0,0,513,0,0,0,0,0,0,0,0,97,6072,1617,142,3
60,0,1,0,0,0,0,0,0,0,0,0,90,0,0,0,00,0,0,0,0,0,0,0,0,108,0,0,0,8,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,10,0,0,0,0,85,353,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,25,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,
0,0,0,0,0,0,3,0,0,0,0,0,0,0,0,00,0,0,0,0,2575,40,0,0,normal
```

Figure 3.6 Bag of system calls feature vector

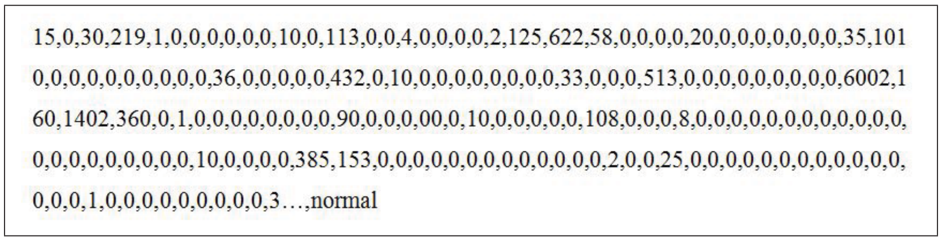
The space complexity of bag of system calls representation is $O(N.|\sum|)$, where N is the number of traces and $|\sum|$ is number of system calls. $|\sum|$ is not a large number. For example, Linux 2.6 has 317 system calls (Inoue and Somayaji, 2007). The cost of this feature vector is much less than successive system calls feature vector representation cost.

3.4.3 *l*_Patterns frequency of system calls feature vector representation

This representation combines properties of the two previous representations. Pattern-frequency representation regards the successive order information of system calls for short patterns, and regards the frequency of those patterns in a trace. Formally, this feature vector representation can be defined as follow: $X_i = \{n_{i,1}, n_{i,2}, \dots, n_{i,|\sum|}\}$, where $n_{i,j}$ is the number of occurrences of a pattern p_j of length l in the trace T_i and r is the number of possible patterns of length l $(r = |\sum^l|)$.

Figure 3.7 illustrates three-pattern frequency representation of a benign application. Each number represents the frequency of pattern of three system calls length. For example, the numbers 15,0,30,219,1 . . . represents the following: the pattern: (open, fstat64, mprotect) is repeated

15 times, the pattern: (close, close, close) is repeated 0 time, the pattern: (mmap2, mprotect, clone) is repeated 30 times, the pattern: (futex, futex, gettimeofday) is repeated 219 times, and the pattern: (access, access, mkdir) is repeated 1 time, and the last attribute in indicates the sequence for benign application.



```
15,0,30,219,1,0,0,0,0,0,10,0,113,0,0,4,0,0,0,0,2,125,622,58,0,0,0,0,20,0,0,0,0,0,0,35,101
0,0,0,0,0,0,0,0,0,0,36,0,0,0,0,0,432,0,10,0,0,0,0,0,0,0,0,33,0,0,0,513,0,0,0,0,0,0,0,0,0,6002,1
60,1402,360,0,1,0,0,0,0,0,0,0,0,90,0,0,0,00,0,10,0,0,0,0,0,0,108,0,0,0,8,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,10,0,0,0,0,385,153,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,25,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,1,0,0,0,0,0,0,0,0,0,3…,normal
```

Figure 3.7   Pattern frequency system call feature vector

The space complexity of l-patterns frequency of system calls representation is $O(N.|\sum^l|)$, where N is the number of traces and $|\sum^l|$ number of possible patterns. For short patterns (l = 2 or 3) and $|\sum| = 317$, l$|\sum^l|$ is not a large number. This representation consumes more space than bag of system calls feature vector and much less than successive system calls feature vector.

3.4.4 Experiments and results

To measure and evaluate the impact of dataset feature vector representations on the classification performance of the classifiers, The following two standard metrics are used:

- Accuracy rate is the rate of correct predictions over the whole dataset.
- False positive (FP) is the quantity of misclassifying malicious behaviors as benign.

The purpose of the machine learning algorithms is to find a classifier C: D $\rightarrow$ {benign, malicious} that maximizes accuracy of right detection and minimizes false detection. Feature vector representation has an impact on the accuracy and false positive rate of the machine learning classifier.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

The dataset consists of a large set of system-call raw traces of benign and malicious applications. The benign applications are the top 100 popular free applications. They are downloaded from the official Android applications store, Google Play (official market of Android applications). The 90 available Android real malwares are download from (blog Contagio: Mobile malware, 2012). All applications are installed and executed on Samsung galaxy S device running Android OS 2.3. The total number of applications in the dataset $N = 190$ (100 benign applications + 90 malwares). The length of sequences varies, it is greatly depending on the applications. The shortest sequence has 10000 system calls, and the longest sequence has 60000 system calls.

The three feature vectors are examined on the following machine learning classifiers: support vector machine (SVM), naïve bayes (NB), logistic regression (LR) and decision tree (DT). These classifiers are obtained from the *WEKA* machine learning visual package (university of waikato, 2012).

Table 3.6 Classification performance of support vector machine classifier.

Feature vector	**Accuracy**	**False positive rate**
Successive system calls	$95,10\%$	$6,10\%$
Bag of system calls	$92,50\%$	$8,50\%$
Two-Pattern frequency	100%	0%
Three-Pattern frequency	100%	0%

Support vector machine (SVM) classifier classifies data by determining a set of support vectors, which are members of the set of training inputs that outline a hyperplane in the feature space (Mukkamala *et al.*, 2002). Table 3.6 represents the support vector machine classifier accuracy and false positive rates using different data representations.

Naïve bayes (NB) classifier is the simplest form of Bayesian network. It is based on Bayes theorem with heavy independence assumptions (Smola and Vishwanathan, 2008). Table 3.7 shows the performance of naïve bayes classifier using different system calls representations.

The three-pattern and two-pattern frequency representation has the best performance (97,10% accuracy rate and 3,10% false positive rate). Bag of system calls representation has the worst performance (91,30% accuracy and 9% false positive rate).

Table 3.7 Classification performance of naïve bayes classifier.

Feature vector	**Accuracy**	**False positive rate**
Successive system calls	94,10%	6,30%
Bag of system calls	91,30%	9%
Two-Pattern frequency	97,10%	3,10%
Three-Pattern frequency	97,10%	3,10%

Logistic regression (LR) classifier is a discriminative model. It is strongly based on the logistic function (Smola et Vishwanathan, 2008). Table 3.8 illustrates the impact of dataset representation on the performance of logistic regression classifier. The three-pattern frequency representation has the highest performance. It has the highest accuracy rate (100%) and the lowest false positive rate (0%). The 2-pattern frequency representation still performs better than the two classical representations with 97,10% accuracy rate and 3,10% false positive rate. Bag of system calls representation has the worst performance (92,30% accuracy and 8% false positive rate).

Table 3.8 Classification performance of logistic regression classifier.

Feature vector	**Accuracy**	**False positive rate**
Successive system calls	96%	4,10%
Bag of system calls	92,30%	8%
Two-Pattern frequency	97,10%	3,10%
Three-Pattern frequency	100%	0%

Decision tree (DT) classifier is built during the training phase. The nodes represent attributes, the edges represent the possible attributes values, and leafs represent the classes. The classification starts by the root node and moves down in the decision tree relative to attribute values till reach leaf (Amor *et al.*, 2004). Table 3.9 demonstrates decision tree classifier influenced by dataset representations. The classifier has better performance with pattern frequency system calls representation than other dataset representations. The three-pattern frequency representation has the best performance with an accuracy rate 98,20% and false positive rate 2,10%. Successive system calls representation has the worst performance with an accuracy rate 92,10% and false positive rate 9,40%.

Table 3.9 Classification performance of decision tree classifier.

Feature vector	**Accuracy**	**False positive rate**
Successive system calls	92,10%	9,40%
Bag of system calls	96%	4,10%
Two-Pattern frequency	97,10%	3,10%
Three-Pattern frequency	98,20%	2,10%

3.4.5 Discussion and conclusion

Feature vector representation has a considerable impact on the performance of machine learning classifiers. It increases the accuracy rate and decreases the false positive rate. The experiment results affirm that the performance of classifiers is improved significantly with same data quantity and type, but with using a different feature vector representation.

Three-pattern frequency feature vector has the best performance of all the experimented classifiers; it reaches the idle performance of SVM classifier and logistic regression classifier (100% accuracy rate and 0% false positive rate). The cost of improving the performance by using dataset representation factor is low relative to other factors: benign behavior model and dataset (size, type).

Because feature vector representation keeps the same dataset type and size, the required memory space is few extra space. The process of preparing feature vector representations is not computationally expensive. Bag of system calls representation has the lowest space complexity and and acceptable accuracy. Successive system calls representation has the highest space complexity, but it does not have the highest performance.

Short pattern system calls representation has extra space complexity than bag of system calls representation, but it significantly improves the accuracy. The frequency property and the successive property of system calls in sequence are important information in the classification process. The pattern frequency property offers advantage by holding these two properties which leads to a better classification process.

CHAPTER 4

CANONICAL BENIGN BEHAVIOR DATABASE APPROACH

4.1 Canonical database

Forrest *et al.* (1996) proposed a new paradigm to detect anomaly for UNIX programs. Their approach is based on monitoring of the system calls generated by a particular process. It builds a database of all unique short system call patterns encountered during the execution of the given process. This database represents the benign behavior of the monitored process. Therefore, we assume that any system call pattern, which is not in the database, may identify an anomalous behavior or a variation in benign behavior. This approach is simple and computationally efficient because it does not have to compute frequency histograms or complex statistics or identify the importance of particular patterns. It just checks the presence or the absence of a pattern. This approach has been used on the UNIX operating system and outperformed numerous approaches (Warrender *et al.*, 1999; Forrest *et al.*, 1996).

The approach suggested by (Forrest *et al.*, 1996). builds a specific benign behavior database for each process of interest. Because of the large number of applications available on Android market, building up a separate database for each application is memory consuming and unrealistic to implement. Thus, our approach is to build a canonical benign behavior database from a limited number of representative applications (Amamra *et al.*, 2014).

This approach can be seen as the last line of defense in an in-depth protection strategy for smartphone systems. In the absence of any specific detection mechanism, our approach may have the last opportunity to detect any malicious activity. If a trusted model describing the benign behavior of a given application is not available, one can use the canonical benign behavior database instead.

The first step is to constitute a canonical benign behavior database composed of the short system call patterns of benign applications. Benign behavior profiles have an important influence on malware detection process. An over-generalized profile would lead to a high rate of false negative alarms (i.e., undetected malware). On the other hand, an under-generalized profile would lead to a high rate of false positive alarms (i.e., benign applications reported as malware). Therefore, benign behavior profile must be consistent and stable.

In order to build a canonical benign behavior database, we have selected the 200 most downloaded free applications from the Google Play (official Android application market). These applications have been installed and executed on a Samsung Galaxy S mobile phone running Android 2.3. To construct the database of patterns of length k, the resulting traces have to go through (i) the filtering and abstraction module described in the previous chapter and (ii) the pattern extraction module to generate the pattern database. This latter module slides a window of length k across the input traces and records all the encountered patterns. The duplicated patterns are ignored. This process is shown in Figure 4.1 with $k = 4$. Different k values are examined to determine the optimal value that ensures compact and efficient anomaly detection system (Amamra *et al.*, 2014).

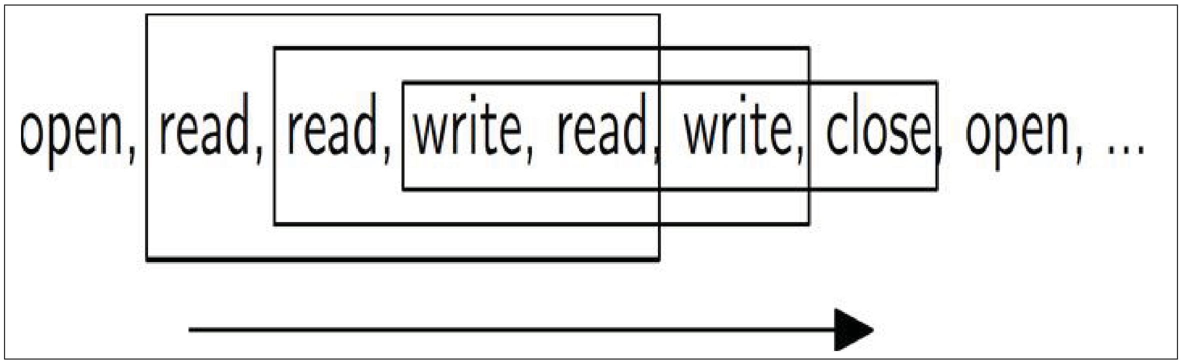


Figure 4.1 Extraction of four-system call patterns with a sliding window

Figure 4.2 illustrates the number of unique patterns as function of the number of considered applications. After training phase using 170 over 200 selected applications, the number of unique patterns converged to 1,786 patterns.

The pattern length is an important factor. It impacts the size of the database, the processing complexity, and the detection capabilities. Short patterns produce a compact database, faster training phase, and less overhead during anomaly detection process. However, short patterns may not represent program behaviors efficiently. This may lead to low accuracy and high rate of false positive alarms. Hence, the minimal pattern length that ensures compact and efficient malware detection must be determined. This value depends on the granularity of kernel calls and varies from one operating system to another and from one version of a given operating system to another. Hofmeyr *et al.* (1998) have determined that six-system call is the best pattern length for UNIX OS. In this work, Different pattern lengths are experimented to determine the optimal value. These experiments are presented in Section 4.4.

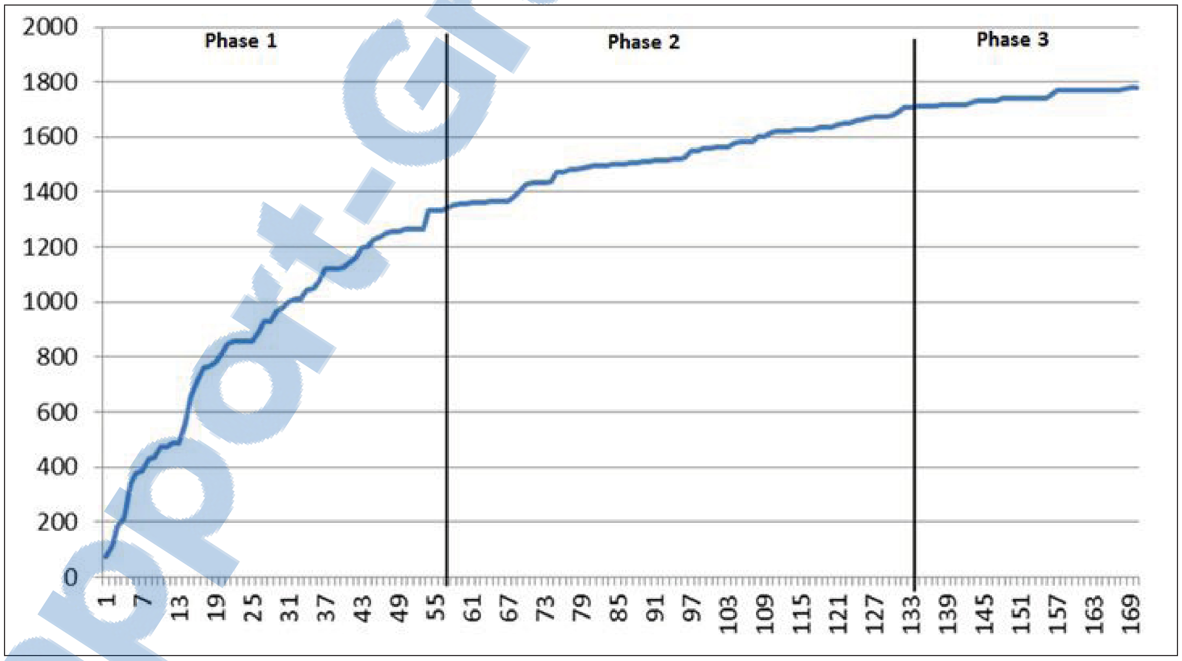


Figure 4.2 The number of four-system call patterns in canonical database benign behavior database as the number of application grows

4.2 Multi-pattern matching algorithms

Once the canonical database has been built, we can use it to detect abnormal behavior of new applications. When an application is monitored, its patterns are compared against the canonical

Rapport-Gratuit.com

database, and the pattern does not exist in the database, can be encountered. Such a pattern is called a mismatch.

Multi-pattern matching algorithms are used to verify the existence of new patterns in canonical database. Therefore, the performance canonical database approach relies on the performance of adopted multi-pattern matching algorithms (Amamra *et al.*, 2012a). Formally, multi-pattern matching problem can be defined as follows:

Considering an input text $T = x_1x_2 \ldots x_n$ of length n and a set $S = \{p_1, p_2, \ldots, p_m\}$ of m keywords. these keywords have different lengths. The problem is to find all occurrences of any of these keywords in the input text T (Kouzinopoulos and Margaritis, 2011).

The naïve solution of this problem is to scan the input text T for each keyword, which requires a total scanning time $n * M$, where M is the sum of keyword lengths. This solution is inefficient, especially with large input texts and large keywords set. In our context, the input text T corresponds the trace of new application and the keywords corresponds the patterns of the canonical benign behavior database.

Multi-pattern matching problem is efficiently solved by various algorithms. In the succeeding text, the mostly known and used ones are explained in details.

4.2.1 Aho-Crosick (AC) algorithm

The Aho-Corasick or simply AC algorithm (Aho and Corasick, 1975; Kelly, 2006) main idea is constructing a finite state automaton from the patterns during a pre-processing phase. The automaton is then used to process an input text string T in a single pass by reading the characters in T and making state transitions or emitting outputs. The automaton machine is based on three functions: *Goto*, *Fail*, and *Output function*. The matching process is straightforward; each transition between two states requires a constant time and only one input character is read. The AC algorithm always has $O(n)$ time complexity regardless of being in best, average or worst case. The space complexity of the AC algorithm depends on the size of that automaton. Consequently, in the worst case the required space is in $O((SumofKeywordsLengths) \times (SizeofAlphabet))$.

4.2.2 Commentz-Walter (CW) algorithm

The Commentz-Walter (CW) algorithm (Commentz-Walter, 1979; Kelly, 2006) idea is similar to the Aho-Cosick algorithm. Constructing the finite state automaton represents the reversed keywords during the pre-processing phase and using a skip table that is similar to the Boyer Moore algorithm (Boyer and Moore, 1977). The matching phase runs in time $n/lmin$, which is the best case. In the case where the matching process keeps incrementing by one, the algorithm runs on $n * lmax$, which is the worst case. $lmin$ is the length of shortest keyword and $lmax$ is the length of the longest one. The space complexity of the algorithm depends mainly on the size of the automaton. In the worst case, the required space is in $O((SumofKeywordsLengths) \times (SizeofAlphabet))$.

4.2.3 Wu-Manber (WM) algorithm

The Wu-Manber (WM) algorithm (Wu *et al.*, 1994; Kelly, 2006) is based on the shifting idea of *Boyer Moore* algorithm. This algorithm scans the input text by blocks of characters of size B, instead of scanning characters one by one. The algorithm is based on three tables built during the pre-processing phase: a SHIFT table, a HASH table, and a PREFIX table. In addition, the space complexity of the algorithm depends on the size of those tables. The algorithm has sub-linear running time complexity in the average case and quadratic in n in the worst case.

4.2.4 Set Backward Oracle Matching (SBOM) algorithm

The SBOM algorithm (Allauzen *et al.*, 2001; Kelly, 2006) is based on an acyclic automaton name factor oracle constructed from the set of reversed patterns. The factor oracle is created during the pre-processing phase. In the matching phase, the algorithm scans the input text by a sliding window of length $lmin$. In this window, the algorithm reads from right to left the

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

longest suffix that labels a path from the initial state. The space complexity of the algorithm is the factor oracle size, and it is at the most $M+1$ nodes. On average, SBOM time complexity is sub-linear in n, but the worst case time complexity is in $O(n*M)$, where M is the sum of the pattern lengths.

4.2.5 Algorithm requirements

Anomaly detection problem on smartphone systems is more specialized than the general patterns matching problem. The choice of a matching algorithm for the anomaly detection problem depends on some specific requirements. Below, we list these requirements compiled from the existing research targeting intrusion detection in general, and a recent contribution targeting intrusion detection specific to mobile devices (Amamra *et al.*, 2012a; Kelly, 2006; Van Lunteren *et al.*, 2006; Zhang *et al.*, 2009):

- *Efficient space complexity*: because a smartphone system has limited memory space, the matching algorithm should consume as less memory as possible. AC algorithm memory requirement is related to the size of the automaton, which is related to the alphabet size $|\sum|$ and the sum of pattern lengths M. In the worst case, the space complexity is in $O(M*|\sum|)$. The space complexity of CW algorithm is based on the automaton and skip functions is in $O(M*|\sum|)+(skipsfunctionrequirement)$. It is obviously more than the AC space complexity. The WM space complexity is related to the space required by the three tables: SHIFT, HASH and PREFIX. The SHIFT table size is in $O(|\sum|^B)$, where B is suffix block length. The HASH table size is the number of SHIFT table elements containing 0. PREFIX table size is in $O(|\sum|^{\bar{B}})$, where $\bar{B}$ is the prefix block size. The space complexity of SBOM algorithm is related to factor oracle for set of keywords. The factor in the worst case has $M+1$ nodes. Then, the space complexity is in $O(M+1)$ + transitions function.
- *Efficient and Stable Performance*: an algorithm has efficient and stable performance only if it always runs on the same time complexity. A common attack on multi-pattern matching algorithms is to force the algorithm to deliberately converge to the worse case complexity,

which is very critical for resources-constrained devices, such as smartphone systems. The AC algorithm time complexity is in $O(n)$ in all cases, which means that AC has stable performance. However, CW, WM and SBOM algorithms run in sub-linear time complexity in the average case and in $O(n^2)$ in the worst case.

- *Scalable*: the multi-pattern matching algorithm should support new patterns and this can be performed with acceptable cost/performance overhead. AC, CW, BSOM and WM are sensitive to the number of patterns because the space complexities are based on the sum of pattern lengths. The four algorithms support large number of patterns in addition to memory cost.

From the aforementioned discussion, the AC algorithm is the best candidate because it has efficient time and space complexities and stable performance which means resistant to the attack that force the algorithm to run to the worse case complexity. Table 4.1 summarizes the algorithms complexities and performance.

Table 4.1 Multi-pattern matching algorithms comparison.

Algorithm	**Time complexity**	**Space complexity**	**Stability**	**Simple to understand and implement**
Aho-Crosick (AC)	$linear$	$O(M \times \sum)$	Yes	Yes
Comments-Walter (CW)	$sub-linear$ or $quadratic$	$O(M \times \lvert\sum\rvert)$ + skip function	No	No
Wu-Manbar (WM)	$sub-linear$ or $quadratic$	$O(\lvert\sum^B\rvert)$ + $O(\lvert\sum^{\bar{B}}\rvert)$ + HASH table size	No	No
SBOM	$sub-linear$ or $quadratic$	$O(M+1)$+ transition function	No	Yes

4.3 Measuring abnormal behavior

When a new application is installed and executed on smartphone system. The system call trace of this application is monitored, a pattern, which does not exist in the canonical database, can be encountered. Such a pattern is called a *mismatch*. In such a case, we should evaluate how abnormal this mismatch is. This can be done by measuring the minimal distance between this mismatch and the patterns in the database. For this purpose, the *Hamming distance* is used. The *Hamming distance* between two patterns p_i and p_j, denoted by $d_H(p_i, p_j)$, is simply given by the number of non-matching positions. Therefore, the *anomaly level* of a mismatch p^* with respect to a database D can be measured as follows:

$$d_{min}(p^*) = \min\{d_H(p^*, p), |p \in D\} \quad > \quad 0 \tag{4.1}$$

This value can be used during the monitoring of a new application to evaluate how abnormal the application is. Different methods have been proposed in the literature. We review these methods and propose a new one.

The first method is the number of encountered mismatches in a trace T. It is the simplest and easiest method to evaluate the anomaly level of an application. Thus, the value is simply given by

$$M(T) \quad = \quad |\{p^* \in T | d_{min}(p^*) > 0\}| \tag{4.2}$$

An application is regarded as abnormal if the number of mismatches exceeds a certain threshold (Forrest *et al.*, 1996; Hofmeyr *et al.*, 1998). However, this value does not take into consideration the anomaly level of the different mismatches. It considers the minimal mismatched patterns as the maximal ones. However, they may not represent the same threat.

Table 4.2 presents the number of mismatches found in the traces of three different applications using four-system-call patterns. Applications 1 and 2 have the same number of mismatches, but these mismatches have different anomaly levels. On the other hand, application 3 has less mismatches than Application 1, but they have higher anomaly levels. Thus, the number of mismatches does not capture the real threat of an application.

Table 4.2 The anomaly levels of the traces based on for four-system call patterns

App number	Nb. of mismatches and their anomaly level				Measures of anomaly level		
	$d_{min}(\cdot) = 4$	$d_{min}(\cdot) = 3$	$d_{min}(\cdot) = 2$	$d_{min}(\cdot) = 1$	$M(T)$	$S_A(T)$	$S_W(T)$
1	1	0	0	199	200	1	1.02
2	100	80	20	0	200	1	3.40
3	0	100	50	20	170	0.75	2.47

Another method to evaluate the anomaly level of an application trace T has been introduced by Hofmeyr et al. (Hofmeyr *et al.*, 1998). This value represents the strength of the abnormal signal in pattern of length k and is defined as follows:

$$S_A(T) \quad = \quad \frac{\max\{d_{min}(p^*)|p^* \in T\}}{k} \tag{4.3}$$

This value is simple and is independent of the trace length. However, this level indicates only the mismatch with the highest anomaly level without giving any clear idea of how other mismatches are distributed.

In Table 4.2, application 1 and application 2 have the same S_A value, but they should not represent the same threat since application 1 has only one pattern with $S_A = 1$ while application 2 has hundred such patterns. Similarly, Application 3 has smaller S_A value than application 1 but its threat should be higher since application 3 has more mismatches with greater anomaly levels.

The number of mismatches and the mismatches strength are two important factors to quantify the anomaly level of a trace T. Therefore, we introduce a weighted anomaly level S_W which takes into consideration both factors (Amamra *et al.*, 2014). It is defined as follows:

$$S_W(T) = \sum_{i=1}^{k} w_i \times p_i(T) \tag{4.4}$$

where $p_i(T)$ is the percentage of mismatches in the trace T having anomaly level i and w_i is the weight associated to a mismatch having anomaly level i.

We simply define $w_i = i$. In such a case, $S_W(T)$ is either 0 or greater or equal to 1. Obviously, this value can be parametrized and different weights could be defined by the user of such a system. For example, w_1 can be set to 0 to reduce too many false positive alarms. At the other end of the spectrum, w_k can be set to 2^k to give much more importance to greater mismatches.

Table 4.2 presents the values of the weighted anomaly level for our three applications. This method seems to be more intuitive as expected. The trace of Application 2 represents a greater deviation from the benign behavior database. Similarly, the trace of Application 1 represents a smaller deviation.

4.4 Experiments and results

The experiments are presented in the following two subsections. In the first one, we evaluate the impact of the filtering and abstraction process on the size of the traces and on the use of the smartphone system resources. In the second one, we evaluate the accuracy of the proposed anomaly-based detection technique. In this case, we have to address two specific questions:

- Does the fact of removing irrelevant system calls have a positive impact on the capability of distinguishing malwares from benign applications?
- Does the approach of using a canonical benign behavior database make any sense as a last line of defense?

4.4.1 Trace size and resource usage

Large traces have negative impacts on the performance of anomaly-based malware detection techniques. Reducing the size of the traces while preserving the characteristics of the application behavior is an important step for improving the performance of these techniques on resource-limited environments. The number of patterns in the benign behavior database essentially determines the efficiency of the anomaly detection approach. Hence, we first examine the impacts of filtering and abstraction process on the number of patterns.

Table 4.3 The reduction of the benign behavior database size

Pattern length k	Nb. of patterns in the raw DataBase	Nb. of patterns in the refined DataBase	Improvement
4	17068	1786	91 %
6	97855	8283	93 %
8	302880	23221	94 %

The patterns are collected from the traces of 170 most downloadable applications as explained in Section 4.1. Table 4.3 shows the number of patterns before and after the filtering and abstraction process. For example, in the case of four-system call patterns, the number of patterns generated from the raw traces is 17068 patterns. This number is reduced to only 1786 patterns after the filtering and abstraction process.

Traditionally, finite-state automaton can be used to retrieve in linear time the rare occurrences of a set of patterns in a given chain as explained in Section 4.2(e.g., Aho-Corasick algorithm (Aho and Corasick, 1975)). A typical application of such an algorithm is an antivirus software looking for known viruses. In a dual approach, a finite-state automaton can easily encode p legitimate k-system call patterns. In such a case, it is possible to retrieve in $O(n)$ time the rare occurrences of unknown patterns (the mismatches) in a given trace composed of n system calls. Such a finite-state automaton can be computed in $O(k \cdot p \cdot |\Sigma|)$ time and space where Σ represents the system-call *alphabet*.

The proposed approach is performed on Samsung Galaxy S mobile phone with 64 MB memory and 1GHz CPU. The results are reported in Table 4.4. As we can see, the anomaly-based malware detection technique based on the analysis of short patterns in program traces is now possible in resource-limited smartphone systems.

Table 4.4 Time and space complexities of the anomaly-based malware detection technique using k-system call patterns

Pattern length	Raw DB		Refined DB	
	CPU usage	Memory usage	CPU usage	Memory usage
4	40%	8MB	4%	4MB
6	55%	22MB	6%	5MB
8	75%	50MB	13%	10MB

4.4.2 Anomaly detection

In this section, we evaluate the impacts of the filtering and abstraction process on the accuracy performance of our anomaly-based detection technique using k-system call patterns. This process is expected to reduce the amount of information irrelevant to describe the application behaviors without any negative impacts on the malware detection capabilities.

The experiments evaluate the malware detection capabilities before and after the filtering and abstraction process. We analyse the system call traces of 50 benign and 50 malicious applications with the canonical benign behavior database described in Section 4.1 and evaluate the anomaly level of these traces.

For reliable evaluation, we use two different methods to evaluate the abnormality of the traces with respect to the corresponding benign behavior database: the anomaly level S_A that is highly recommended by (Hofmeyr *et al.*, 1998), and our proposed weighted anomaly level S_W (Amamra *et al.*, 2014).

The experiments have been done for three different pattern lengths ($k = 4, 6$ and 8) in order to evaluate the performance of our malware detection approach with respect to this parameter.

4.4.2.1 The accuracy evaluation

In order to evaluate the accuracy of the anomaly-based detection technique based on the *k*-system call patterns, we use the following measures: the *false positive* (FP) rate, the *false negative* (FN) rate and the *accuracy*. These measures are defined formally as follows:

- FN is the percentage of misclassified malwares as benign applications.
- FP is the percentage of misclassified benign applications as malwares.
- The accuracy is the percentage of correct predictions over all data i.e.

$$\frac{\text{true benign applications} + \text{true malwares}}{\text{number of evaluated applications}}.$$

4.4.2.2 Anomaly detection without using the filtering and abstraction process

For the first experiment, we evaluate the accuracy performance of our malware detection technique using the raw traces and the raw canonical benign behavior database composed of four-system call patterns. 50 benign applications and 50 malicious applications have been used for this test.

Figure 4.3 presents the distribution of the anomaly level S_A values. These values can be clearly separated with a threshold set at 0.5 – i.e.,the minimum *Hamming distance* between any unseen pattern in a given trace and any pattern in the database is at least two. Unfortunately, depending whether the limit cases are classified as benign or malicious applications, either four benign applications or three malicious applications are faultily identified. As we can expect, the false positive alarms are rather annoying for the users and must be reduced as much as possible.

The test has been repeated with the weighted anomaly level S_W introduced in this section. Figure 4.4 presents the distribution of the weighted anomaly level S_W values. Even if these values can be clearly separated, it is harder to set the threshold. S_W can take any value

Rapport-Gratuit.com

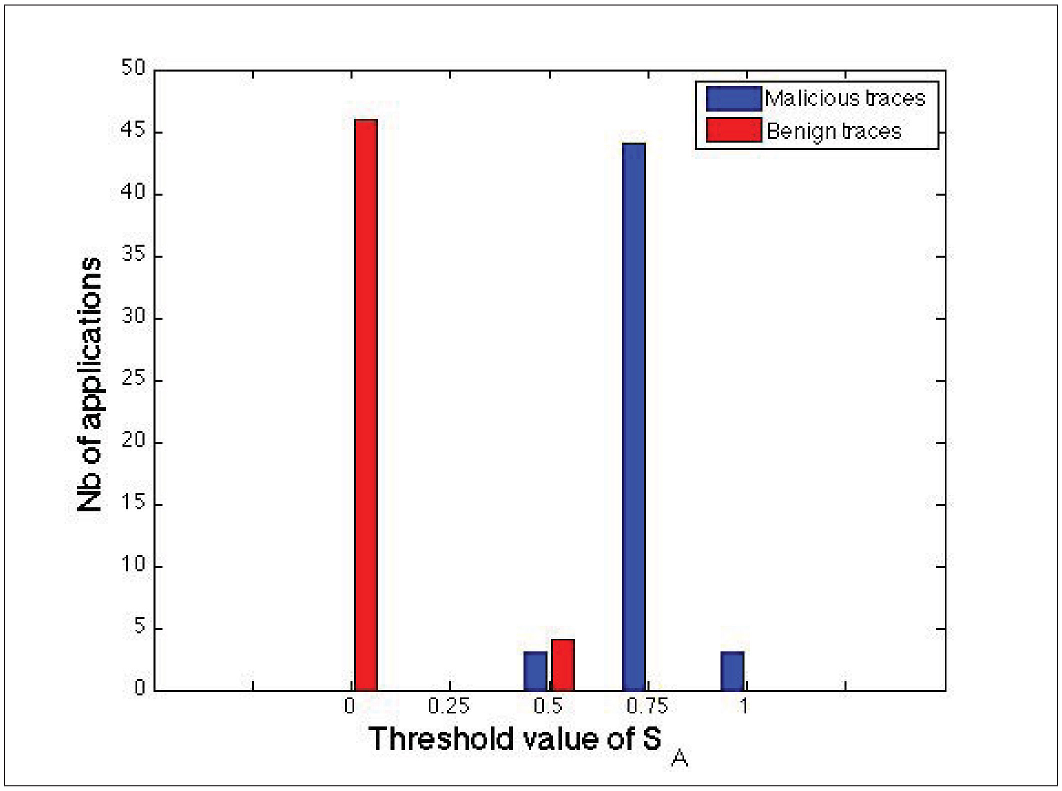


Figure 4.3 The distribution of the S_A anomaly level values for the raw traces of benign and malicious applications (using 4-system call patterns)

greater or equal to 1, except for the value 0 in the perfect cases without any discovered mismatch. Hence, the figure shows the distribution of the values using the following intervals: $0, [1 \cdots 1.1), [1.1 \cdots 1.2) \cdots [1.5 \cdots +\infty)$.

To reduce the false positive alarm rate as much as possible without accepting too many malicious applications, the threshold has been set to 1.2. This misclassifies two benign applications and eight malicious ones.

Tables 4.5 and 4.6 summarize the accuracy performance of our malware detection technique using the raw traces and the raw database.

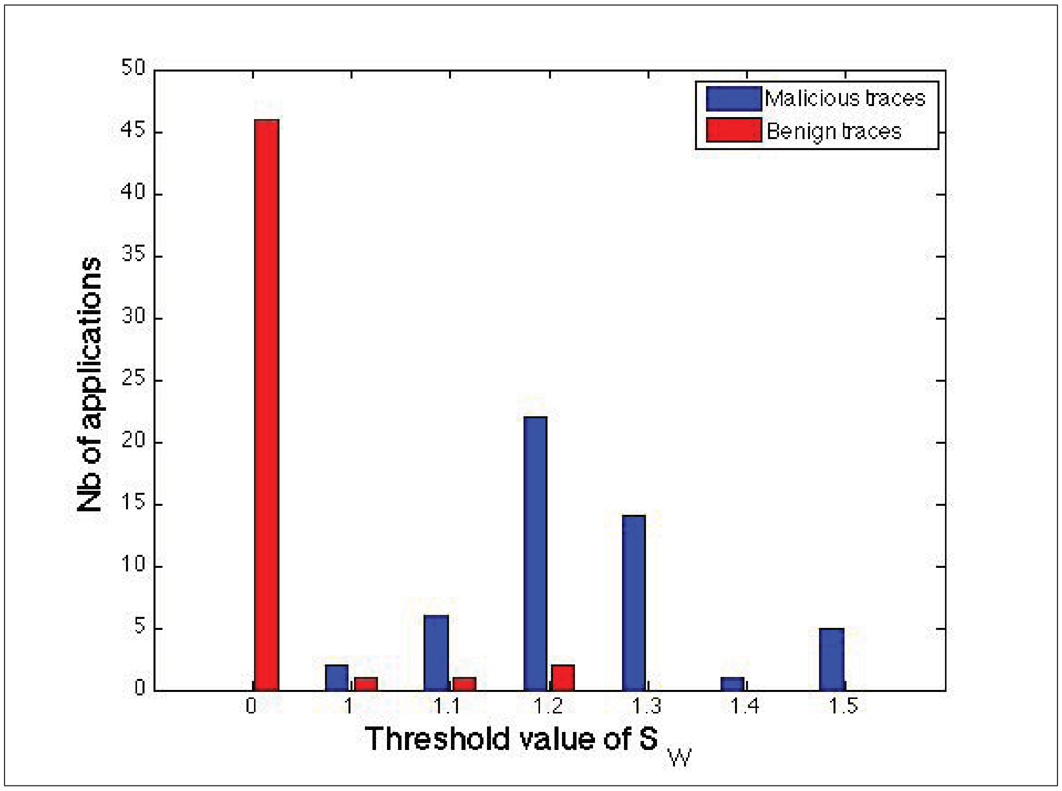


Figure 4.4 The distribution of the S_W weighted anomaly level values for the raw traces of benign and malicious applications (using 4-system call patterns)

Table 4.5 The accuracy performance of malware detection technique using the raw traces and the S_A anomaly level

Four-system call patterns				Six-system call patterns				Eight-system call patterns			
S_A	FP	FN	Accuracy	S_A	FP	FN	Accuracy	S_A	FP	FN	Accuracy
0.25	8%	0%	96%	0.33	4%	0%	98%	0.25	0%	0%	100%
0.5	0%	6%	97%	0.5	0%	8%	96%	0.5	0%	12%	94%

Table 4.6 The accuracy performance of malware detection technique using the raw traces and the S_W weighted anomaly level

Four-system call patterns				Six-system call patterns				Eight-system call patterns			
S_W	FP	FN	Accuracy	S_W	FP	FN	Accuracy	S_W	FP	FN	Accuracy
< 1.1	6%	4%	95%	< 1.1	8%	0%	95%	< 1.3	8%	4%	96%
< 1.2	4%	16%	90%	< 1.2	4%	2%	97%	< 1.4	6%	8%	93%

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

4.4.2.3 Anomaly detection using the filtering and abstraction process

In this part of the experiment, we add the filtering and abstraction process to the malware detection technique. We evaluate the accuracy performance of our approach using the same applications and the same anomaly levels S_A and S_W.

Figure 4.5 presents the distribution of the anomaly level S_A values. These values can be perfectly separated in this case. Therefore, refining the traces (and a fortiori the database) allows to select a higher and therefore stricter threshold of 0.75, which should reduce the crucial false positive alarm rate.

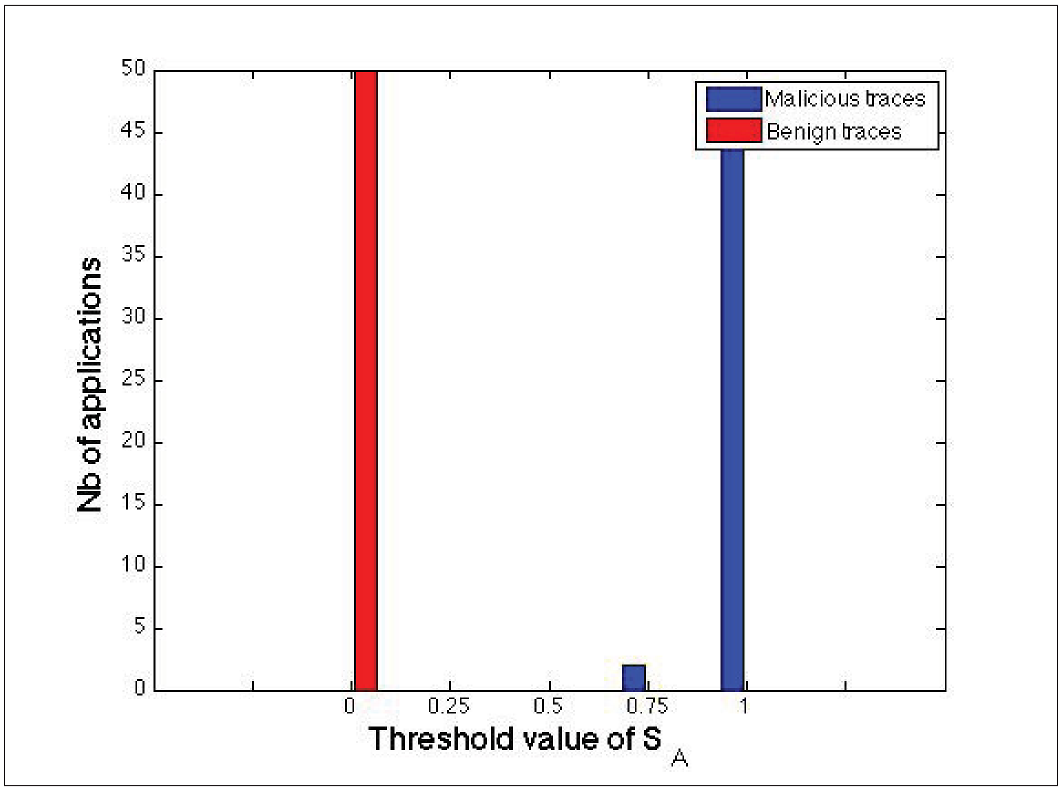


Figure 4.5 The distribution of the S_A anomaly level values for the refined traces of benign and malicious applications (using 4-system call patterns)

Finally, we evaluate the accuracy performance of the approach using the weighted anomaly level S_W. Figure 4.6 presents the distribution of anomaly level S_W values. No mismatch has appeared in the benign application traces. However, setting the threshold too low may be dangerous. Setting the threshold to 1.2 as in the previous section still gives a perfect classification. Thus, it should be more prudent to select a stricter (and more conservative) threshold to avoid any eventual outlier in the benign applications.

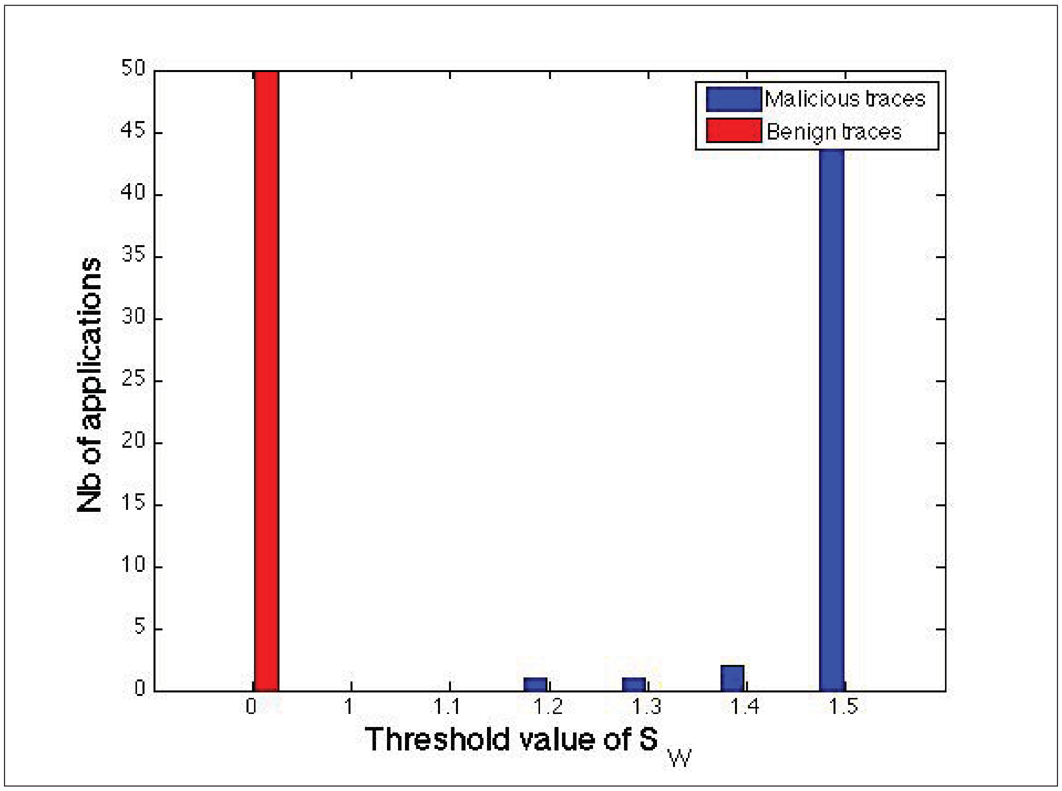


Figure 4.6 The distribution of the S_W weighted anomaly level values for the refined traces of benign and malicious applications (using 4-system call patterns)

For practical purpose, the threshold can be predetermined by the security administrator of the system or by the user according to his need through the threshold parameter in the system setting.

Tables 4.7 and 4.8 summarize the accuracy performance of our malware detection technique using the filtering and abstraction process to refine the traces.

Table 4.7 The accuracy performance of malware detection technique using the refined traces and the S_A anomaly level

4-system call patterns				6-system call patterns				8-system call patterns			
S_A	FP	FN	Accuracy	S_A	FP	FN	Accuracy	S_A	FP	FN	Accuracy
0.75	0%	0%	100%	0.50	0%	0%	100%	0.50	0%	0%	100%
1.00	0%	4%	98%	0.66	0%	72%	64%	0.625	0%	64%	68%

Table 4.8 The accuracy performance of malware detection technique using the refined traces and the S_W weighted anomaly level

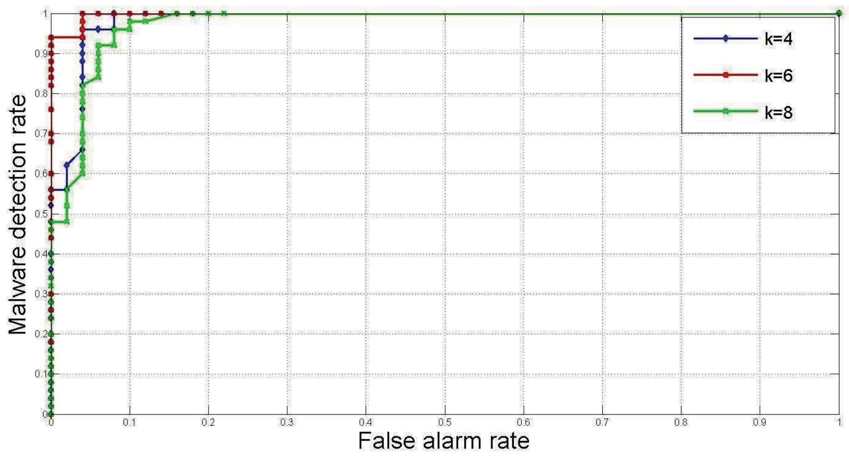
4-system call patterns				6-system call patterns				8-system call patterns			
S_W	FP	FN	Accuracy	S_W	FP	FN	Accuracy	S_W	FP	FN	Accuracy
< 1.2	0%	0%	100%	< 1.2	0%	0%	100%	< 1.2	6%	2%	96%
< 1.4	0%	4%	98%	< 1.4	0%	2%	99%	< 1.4	0%	4%	98%

For fair estimation and comparison of the approach of using canonical benign behavior database, ROC (Receiver Operating Characteristics) curves are constructed for all proposed schemes and are illustrated in Figure 4.7. A ROC curve shows trade-offs between the malware detection rate and false alarm rate when different threshold are used. It is commonly used to visualize and estimate the accuracy performance of detectors Fawcett (2006). It is apparent in Figure 4.7 that the proposed approach is a promising anomaly malware detection technique where all the curves are bulged further outward to the perfect point (0,1).

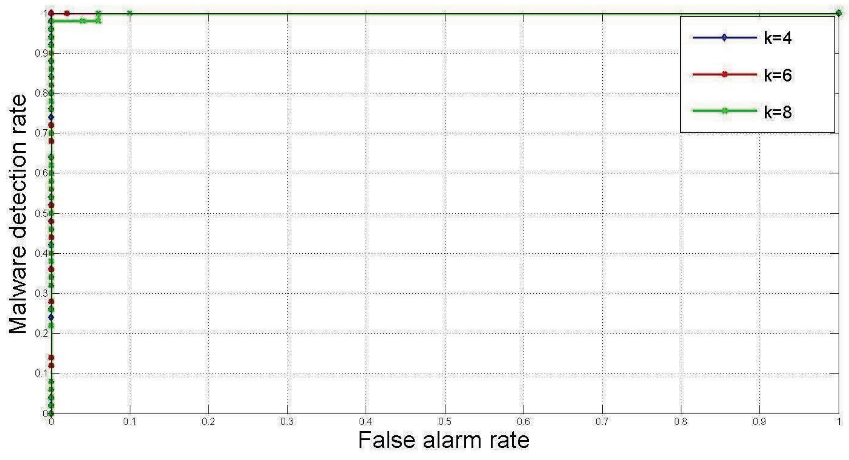
Figure 4.7(a) illustrates the accuracy performance of the approach without the filtering and abstraction process. The approach with six-system call patterns outperform the others. With the filtering and abstraction process, The approach with four-system calls and six-system call patterns reached the perfect curve as illustrated in of Figure 4.7(b). However, the approach with four-system call patterns has less time and space complexities.

4.5 Discussion and conclusion

Our general objective was to investigate whether it was possible to implement on resource-limited smartphone systems a malware detection technique based on the analysis of system



(a) ROC curves of raw normal behavior database



(b) ROC curves of refined normal behavior database

Figure 4.7 The ROC curves of the anomaly malware detection using the canonical benign behavior database appraoch

calls (Forrest *et al.*, 1996, 2008; Hofmeyr *et al.*, 1998). To achieve this objective, two different goals had to be addressed.

Our first goal was to determine whether it was possible to reduce the amount of resources needed to implement the proposed approach. We have presented a filtering and abstraction process, which considers only a limited number of system calls. Such a process significantly reduces the size of the database describing the benign behavior of the applications. Our experimentations clearly show that smartphone systems can find out in real time any abnormal pattern of system calls. In fact, they can retrieve these m mismatches in $O(n+m \cdot k \cdot p)$ time, where n is the size of the trace and p the number of distinct legitimate k system-call patterns. Since m is relatively small for any benign application (according to our empirical results), such a naïve approach is acceptable.

Unpredictably, refining the traces (and therefore loosing some information) does not impact significantly the capability to distinguish between benign and malicious applications. One explanation is the fact that irrelevant system calls are discarded and it may simpler to discover the real intentions of the applications.

Our second objective was to evaluate the approach of using a canonical benign behavior database to distinguish between benign and malicious applications. This database is simply built with the most popular benign applications found on Android market. Obviously, such an approach cannot be as perfect as building application-specific models. However, this latter approach is unrealistic – just too many applications! In an in-depth protection strategy, we need a last line of defense when no application-specific approach exists. In such cases, we want to reduce the false positive alarms as much as possible to avoid annoying the users and to catch as much as possible malwares – we do not have to be perfect, simply good!

Further analysis would have to be done – eventually with real users in real deployment setups. It may be possible that few canonical benign behaviors have to be used – depending on a classification of the applications. However, our first experimentations are promising.

In conclusion, our filtering and abstraction process allows us to propose a lightweight technique, which can be used as the last line of defense of an in-depth protection strategy for smartphone systems. It would be interesting to evaluate whether this process could be used to implement more advanced malware detection techniques based on machine learning algorithms.

CHAPTER 5

MACHINE LEARNING CLASSIFIERS

5.1 Generative vs discriminative classifier

Machine learning is an active research field, which develops systems learning how to distinguish between different classes of data. In their initial training phases, these systems establish the models characterizing the data. These models are simply defined by their parameters (Alpaydin, 2004). Generally, there are two main categories of machine learning classifiers: *generative classifiers* and *discriminative classifiers* (Bishop *et al.*, 1995; Ulusoy and Bishop, 2006).

To clarify the difference between the two categories of classifiers, consider the following simple scenario:

Suppose an Android application a, which is described by a feature vector x, belongs to either one of the two classes $C = \{\text{benign}, \text{ malicious}\}$. From a probabilistic perspective, the objective of a classifier is to find the conditional distributions $\Pr(c|x)$, for $c \in C$, in order to predict whether an application a is benign or malicious.

The *generative one-class classifiers* capture the generation process of x by modelling the joint distribution $\Pr(x, c)$ as a parametric model, and then subsequently use this joint distribution to infer the conditional distribution $\Pr(c|x)$ in order to predict whether or not a new application belongs to a given class (Ulusoy and Bishop, 2006; Y. Ng and Mihael I, 2002). In such a case, the training phase just needs the training data from the given class – in our setting, the benign applications. This approach is particularly interesting since it may be hard to find instances of malicious applications.

By contrast, the *discriminative multi-class classifiers* use parametric models to represent the conditional distribution $\Pr(c|x)$. The parameters of a model are established during a training phase on a set of labelled data (x, c). In such a case, these data must contain data from each

Rapport-Gratuit.com

class. The resulting model can be used later on to predict which class a new application most likely belongs to (Ulusoy and Bishop, 2006).

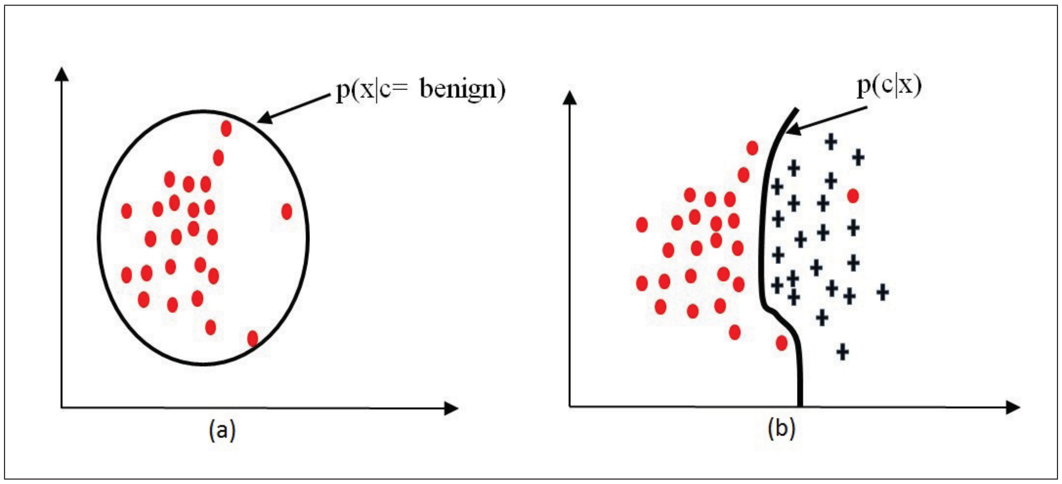


Figure 5.1 A discriminative binary classifier vs. a generative one-class classifier

Figure 5.1 illustrates the difference between (a) a *generative one-class classifier* and (b) a *discriminative two-class (or binary) classifier*. The *generative one-class classifier* focuses on understanding the basic description of the class composed of the benign applications, whereas the *discriminative binary classifier* tries to model the boundaries between benign and malicious applications.

Both approaches have some advantages and disadvantages. The *discriminative classifiers* are very fast at making the prediction for new applications, and they are quite accurate (Ulusoy and Bishop, 2006; Tu, 2007). On the other hand, the *generative classifiers* can treat partially labelled or/and unlabelled data. They can readily treat variances of the models, whereas discriminative models should see all possible variances during training. Generative classifiers use only a single class data for the training process (Tu, 2007). Unfortunately, such approaches often have limited performance accuracies for many applications (Gong, 2008; Yu *et al.*, 2008).

In this work, we analyze two classical classifiers, one from each category. Support vector machine (SVM) classifier is selected to represent discriminative category and *k*-means classifier is selected to represent generative category.

5.1.1 The support vector machine (SVM) classifier

Among the different discriminative classifiers, the support vector machine (SVM) classifier is one of the successful machine learning classifiers for the anomaly detection problem. The two-class SVM learning has relatively fast processing and high detection performance (Shon and Moon, 2007).

Support vector machine classifier tries to find a hyperplane separating the points representing the benign and the malicious applications of the training set. In such a case, an optimal solution would be defined by a pair of parallel hyperplanes delimiting the widest zone (with respect to the orthogonal Euclidean distance) separating the given points.

Unfortunately, such a perfect separation hyperplane does not always exist. In such a case, the constraints should be relaxed and some points may be misclassified. The soft-margin SVM was proposed by Cortes and Vapnik (1995) to solve this problem efficiently. It introduces the slack variables ξ_i and the penalty of cost $C \cdot \xi_i$ for the misclassified points. The slack variables ξ_i determine whether the corresponding points are misclassified or not – i.e., being on the wrong side of the separation hyperplane. If $0 < \xi_i \leq 1$, the given point is too close to the separating hyperplane. If $\xi_i > 1$, the point is simply on the wrong side. Hence, the formulation of soft-margin SVM optimization problem is given by:

$$\min \frac{||w||^2}{2} + C \cdot \sum_{i=1}^{n} \xi_i \tag{5.1}$$

subject to the constraints $y_i(wx_i + b) \geq 1 - \xi_i$ and $\xi_i \geq 0$, for $i = 1, \cdots, n$.

The function $|| \; ||^2$ denotes the Euclidean distance in the Euclidean space R^d. The training data $(x_i,\, y_i)$ is used to compute the normal vector to the hyberplane w and threshold b. C is a

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

parameter to be chosen by the user, it controls the trade-off between maximizing the margin and minimizing the training error. A small C allows the margin constraints to be easily violated. This is equivalent to create wide margin around the decision boundary. On other hand, a large C makes the margin constraints hard to be violated (Chapelle *et al.*, 2002). Figure 5.2 illustrates the soft-margin SVM separating between two different classes in the 2-dimensional Euclidean plane.

Once the support vector have been trained, it can be used to determine the class of given test application x. the decision function is given by:

$$f(x) = sgn(w.x + b) \tag{5.2}$$

The first algorithm used to solve the above quadratic optimization problem is "chunking" algorithm. The time complexity of the "chunking" algorithm is in $O(n^3)$ and the space complexity is in $O(n^2)$ (Tsang *et al.*, 2005), where n is the training set size. An alternative algorithm has been proposed which is Sequential Minimal Optimization (SMO) learning algorithm (Platt *et al.*, 1999). SMO has $O(n^2)$ time complexity and $O(n)$ space complexity (Platt *et al.*, 1999). In classification phase, it simply requires the time complexity to evaluate the equation (5.2) which is $O(m)$, where m is the size of test set. In our experiments, we have used SMO algorithm.

5.1.2 The *k*-means classifier

One-class classification differs from the multi-class classification in its fundamental aspect. In one-class classification, only the information of the given class is used during the training phase. Therefore, one-class classification could be used when the data from the other classes are difficult to obtain (e. g., malicious applications).

The *k*-means classifier has been selected as a generative one-class classifier due to its simplicity, stability, and efficiency. It converges rapidly to either local or global minimum (Jain,

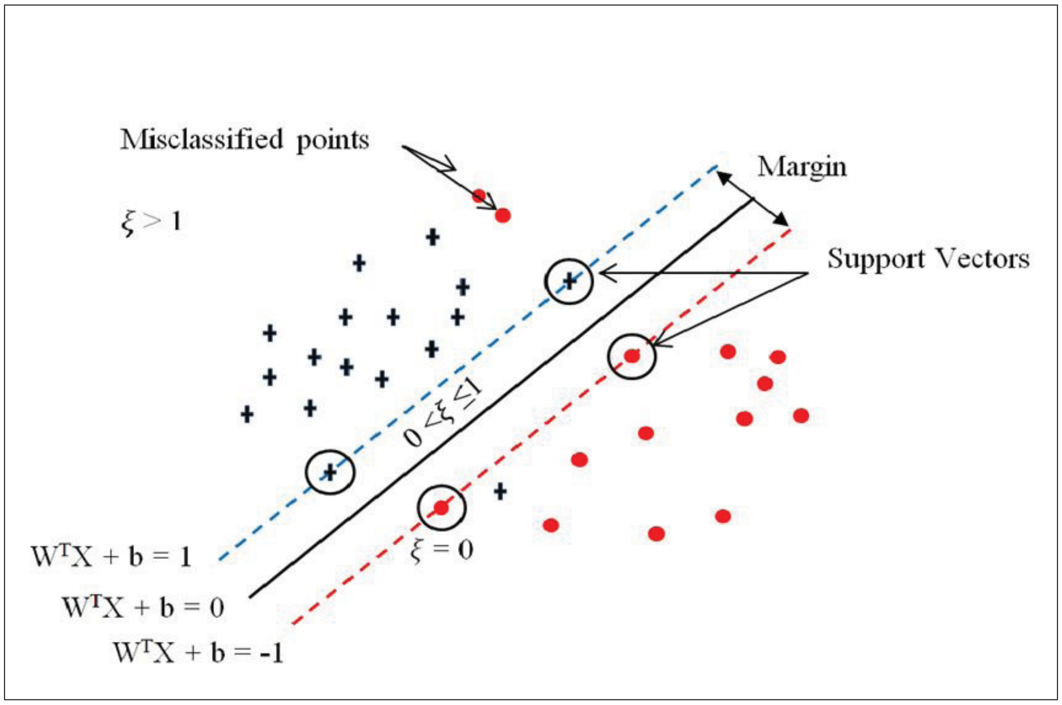


Figure 5.2 Soft margin Support vector machine

2010). This classifier is one of the simplest unsupervised classifiers solving the clustering problem. The *k*-means classifier assumes that the data of the given class can be clustered around a set of centroids points $c_j,\ j = 1, 2, \cdots k$. The number of centroids should be determined beforehand (Chiang and Mirkin, 2010; Sugar and James, 2003). Figure 5.3 illustrates *k*-means classifier with two centroids.

The locations of the centroids can be optimized during the training phase by minimizing the following cost function:

$$f(S, c_1, \cdots, c_k) = \sum_{j}^{k} \sum_{x_i \in S_j} \|x_i - c_j\|^2 \tag{5.3}$$

where the set S_j partition the dataset S.

This problem is known to be NP-hard.Thereofre, *k*-means algorithms are approximation algorithms, which generally converge to local optima (Jain, 2010).

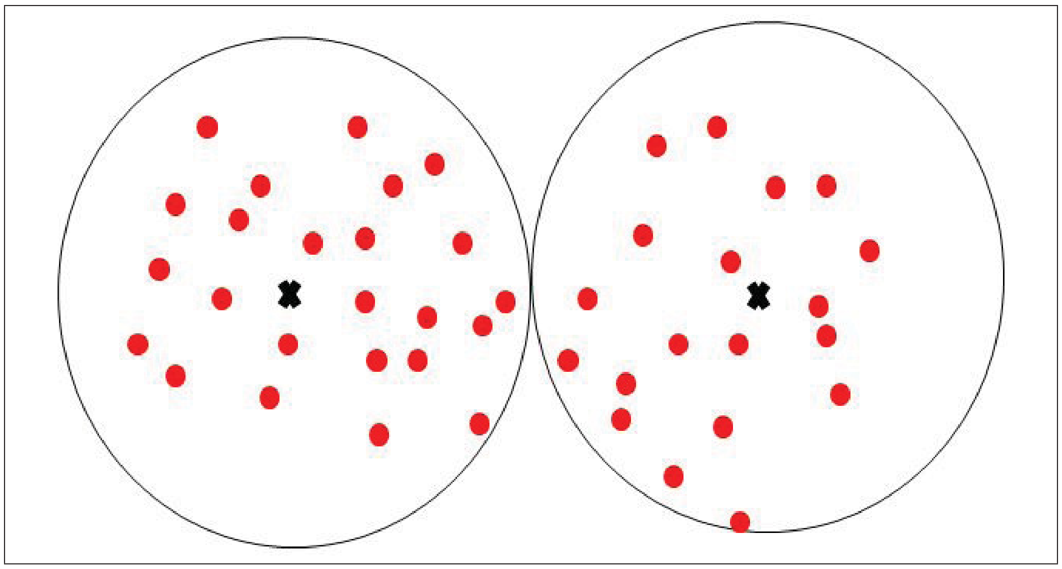


Figure 5.3 One-class *k*-means classifier with two centroids

After the training phase, the target data is organized in clusters and each cluster has centroid and radius. The radius represents the distance between the centroid and the most distant point of the same cluster and it determines the bounds of the normality.

During the classification phase, the membership function is calculated as follow:

$$f(x) = \min_j \|x - c_j\|^2 \tag{5.4}$$

The result $f(x)$ is compared with a threshold T. If it is less than or equal to the threshold, x would be assumed to be a member of the class. Otherwise, it would not.

The two main ways used for setting the threshold T value are:

- The threshold is set such that the data located in the far tails of the distribution are rejected. For example, Giacinto *et al.* (2005) and Ratle *et al.* (2007) had set target rejection rate to 5%.

- The threshold is set equal to the radius of the nearest centroid (Hodge and Austin, 2004). In our experiments, we adopted this threshold settings.

The time complexity of the training phase is in $O(n \times d \times k \times l)$ where n is the number of training vectors, d is the dimension of the vectors representing the data points, k is the number of centroids and l is the number of iterations until the cost function $f(S, c_1, \cdots, c_k)$ no longer changes. The space complexity is in $O((n+k) \times d)$ (Tan *et al.*, 2005). The classification/detection phase is much faster. Its time complexity is in $O(d \times k)$, and its space complexity is simply in $O(k)$.

5.1.3 Experiments and results

To evaluate the accuracy performance of the classifiers, the confusion matrix (Fawcett, 2006) is used as presented in Table 5.1.

Table 5.1 The confusion matrix.

		Actual class	
		Malware	Benign
Predicted class	Malware	True Positive (TP)	False Positive (FP)
	Benign	False Negative (FN)	True Negative (TN)

The *true positive rate* (TPR) is the proportion of malicious applications that are correctly classified. The *true negative rate* (TNR) is the proportion of the benign applications that are correctly classified. Whereas, the *false positive rate* (FPR) is the proportion of misclassified benign applications as malicious applications and the *false negative rate* (FNR) is the proportion of the undetected malicious applications. Traditionally, the *accuracy* (Fawcett, 2006) evaluates the correctness and the overall detection rate of the classifiers. It measures the overall detection rate.

5.1.3.1 The experiment protocol

The classical f-fold cross validation test (Dietterich, 2000; Kohavi *et al.*, 1995) is usually employed for credible evaluation of classifiers. This test splits the data set into f subsets of equal size. Once the data set has been partitioned, one subset is selected and the classifier is trained on the other $f-1$ subsets. The classifier then tries to identify the data from the selected subset. Since the data has been labelled beforehand, the accuracy of the classifier on that subset can be evaluated. This experiment is repeated for each of the f subsets. The average μ and the standard deviation σ of the accuracy values are computed, which represent the average performance of the classifier. The advantage of such a test is that all instances of the data are eventually used for both the training phase and the classifying/detecting phase. Thus, the experiment results should be more representative of the real classifier accuracy.

Confidence interval (Dietterich, 2000) illustrates the reliability of an estimate. It is given by the following formula:

$$\mu \pm z \times \frac{\sigma}{\sqrt{f}} \tag{5.5}$$

The confidence level parameter z is usually given by the cumulative normal distribution function. For example, $z = 1,96$ for a confidence interval of 95%.

To determine the optimal parameters of the classifiers, the dataset is divided into two sets: validation set and test set. The validation set is used to validate the model, where different parameters are tried and the model of the best accuracy performance is selected. In the case of the k-means classifier, the number of centroids k impacts the complexity and classification process of the classifier. Therefore, determining the optimal k is very important. In the case of SVM classifier, the C parameter controls the tradeoff between margin maximization and error minimization. In addition, it impacts the classifier complexity.

After determining the best classifier models, the test set is used to test this model. 5-fold cross validation is used and each experiment is repeated 10 times and the reported results are the mean of the 10 repetition.

5.1.3.2 The dataset for experiments

The dataset used to evaluate the classifiers is composed of the system call traces of 100 benign applications and 100 malicious applications. The benign applications are the most downloaded applications from Android official market (Google Play). The malicious applications are real malware that are downloaded from (blog Contagio: Mobile malware, 2012).

In our experiment, the version of the Android operating system is 2.3 based on Linux 2.6. It has 317 different system calls. On the other hand, the filtered-and-abstraction process reduces this number to 125. This should significantly improve the performances of the different classifiers. The dataset is divided into 10-folds. Each fold consists of 10 vectors representing benign applications and 10 vectors representing malicious applications. The first 5-folds are used to fix the parameters of models (validation process) and the second 5-folds are used to test the model performance (test process).

5.1.3.3 The results

As mentioned in Section 5.1, two classifiers have been chosen for the experiments: the k-means and the SVM classifiers. The selected implementations of these classifiers come from the WEKA machine learning visual package (university of waikato, 2012).

The *k*-means classifier

The k-means classifier is a generic one-class classifier. Hence, it is trained only with the benign applications, and it is tested with the benign and the malicious applications. The performance of the k-means classifier depends greatly on the number of clusters k. To determine the optimal k value for our dataset, a simple exhaustive search using 5-folds cross validation is used. Different values of k are tried, and the model with best accuracy is picked. Figure 5.4 illustrates values of k and the accuracy corresponds to each model. For raw traces, $k = 5$ is the optimal value, whereas the optimal one for refined traces is $k = 2$.

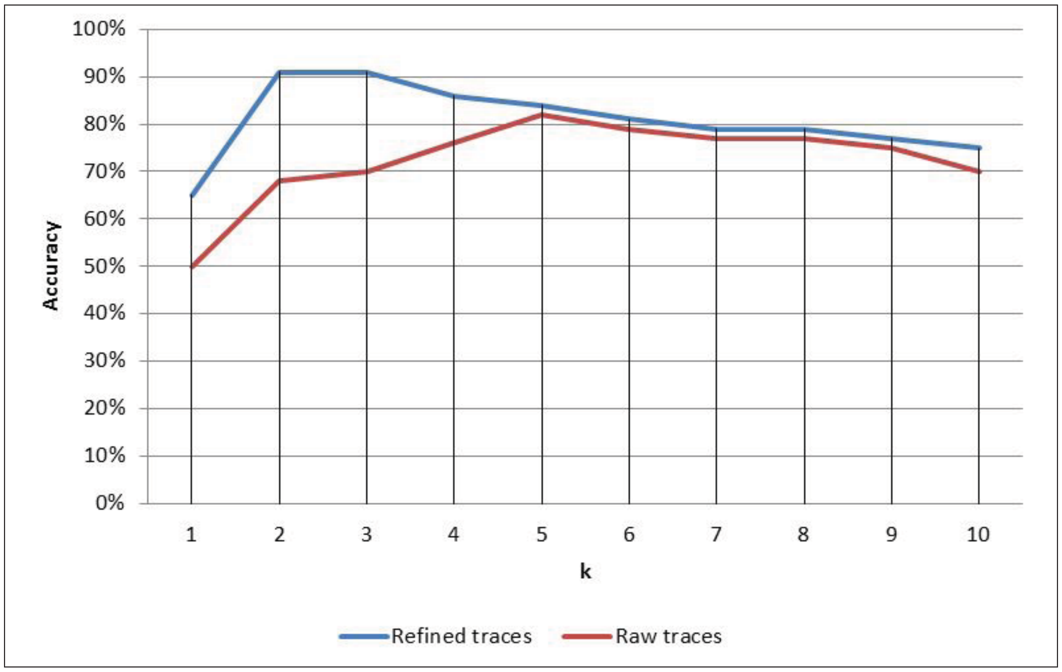


Figure 5.4 k-means accuracy vs. the value of k for raw and refined traces

Table 5.2 and Table 5.3 report the experiment results of the validated models of the classifier on the raw and refined traces respectively. On the raw traces, the malware detection rate (TP) is around 81% and the false positive alarm rate (FP) is around 22.5%. Unfortunately, the overall accuracy performance is relatively poor at 80%. The prediction performance of the classifier can be significantly improved if the refined traces are used instead. Around 90% of malicious applications are detected, while only 9% of the benign applications are identified as malicious applications. The overall accuracy performance of the classifier is relatively good at 90%.

Table 5.2 k-means accuracy performance using the raw traces.

		Actual class	
		Malware	Benign
Predicted class	Malware	$81,80\% \pm 2,35\%$	$22,50\% \pm 2,80\%$
	Benign	$18,20\% \pm 2,35\%$	$77,50\% \pm 2,80\%$
		Accuracy = $79,65\% \pm 2,53\%$	

Rapport-Gratuit.com

Table 5.3 k-means accuracy performance using the refined traces.

		Actual class	
		Malware	Benign
Predicted class	Malware	$90,70\% \pm 1,45\%$	$8,90\% \pm 1,65\%$
	Benign	$9,30\% \pm 1,45\%$	$91,10\% \pm 1,65\%$
		Accuracy = $90,90\% \pm 1,53\%$	

The SVM classifier

The performance of the SVM classifier is largely dependent on the parameter C. It is important to find the optimal fitting value without over fitting the model. Increasing the value of C increases the misclassification cost and produces a more accurate classifier at the risk of losing the generalization properties. Furthermore, a large value of C usually increases the training time. On the other hand, decreasing false classified instances improves accuracy performance.

To determine the optimal C value for our dataset, "grid-search" using 5-folds cross validation has been used (Chang and Lin, 2011). Various values of C parameter has been tried, and the model with best accuracy has been selected.

Figure 5.5 illustrates the SVM accuracy vs different C values. The optimal value of C for the raw traces is 35, and the optimal one for the refined traces is 25.

Table 5.4 reports the accuracy performance of the two-class SVM classifier on the raw traces of the optimal models. The malware detection rate is relatively high, but unfortunately the false positive and negative alarm rates are not negligible. Nevertheless, the two-class SVM classifier has a good overall accuracy rate of 92%. The performance is even better on the refined traces as illustrated in Table 5.5. The results are simply perfect on the refined traces.

5.1.3.4 Resources usage

To evaluate the resource usage and the performance of each classifier, these classifiers are implemented and executed on Google Nexus S smartphone device. Table 5.6 illustrates the

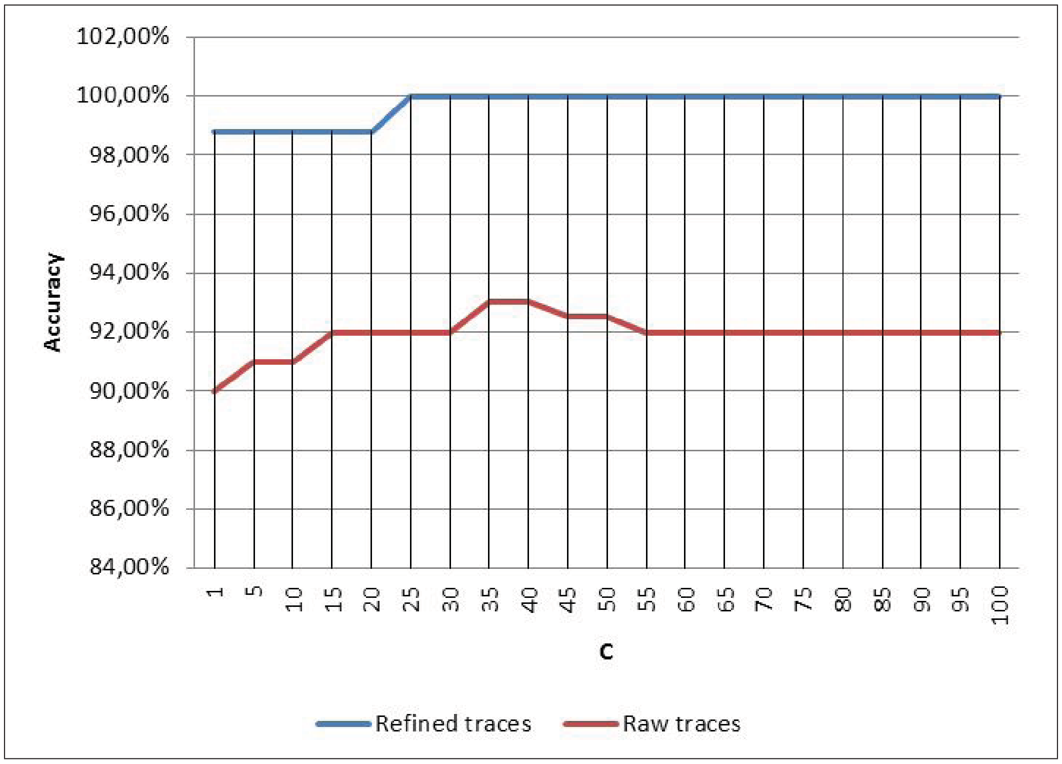


Figure 5.5 SVM accuracy vs C parameter

Table 5.4 SVM classification results using the raw traces.

		Actual class	
		Malware	Benign
Predicted class	Malware	$92,30\% \pm 1,09\%$	$7,70\% \pm 1,40\%$
	Benign	$7,66\% \pm 1,09\%$	$92,50\% \pm 1,40\%$
		Accuracy = $92,32\% \pm 1,15\%$	

used space of the both classifiers is very close in the case of raw traces and they used the same space in refined traces. From the execution time point of view, SVM classifier is faster than the

Table 5.5 SVM classification results using the refined traces.

		Actual class	
		Malware	Benign
Predicted class	Malware	100%	0%
	Benign	0%	100%
		Accuracy = 100%	

k-means classifier in both cases. The refined traces improve the execution time of the *k*-means classifier by 49,69%, and the SVM classifier by 43,27%.

Table 5.6 Time and space complexities of SVM and k-means classifiers.

	Raw traces		Refined traces	
	Space usage (MB)	Time (ms)	Space usage (MB)	Time (ms)
k-means	11	10619	9	5277
SVM	12	594	9	257

5.2 Hybrid classifier

Machine learning classifiers are extensively used for anomaly detection problem. For example, Bose *et al.* (2008) proposed a framework to detect smartphone malwares based on the SVM classifier. Shabtai *et al.* (2012) proposed a framework that applies different machine learning classifiers (e.g., *k*-means, logistic regression, histograms, decision tree, bayesian networks and naïve bayes) to detect malwares of smartphone system. The benign behaviors are constructed by training a single classifier. The single classifiers yield good classification results. However, anomaly-based detection techniques still need more investigations and improvements. In this chapter, we investigate and evaluate the possibility of enhancing the performance of anomaly-based detection by using hybrid machine learning classifiers. A hybrid classifier combines two or more different single classifiers in order to improve the classification process.

A hybrid classifier is a set of classifiers whose single predictions are combined in some way to classify new samples. They use the output of the classifiers without considering details of their implementation. Hybrid classifiers produce higher prediction accuracy at the meta-level (Amamra *et al.*, 2012c; Džeroski and Ženko, 2004; Sigletos *et al.*, 2005). In this chapter, we examine the approach proposed by Wolpert (1992). It is an old approach for combining multiple classifiers, known as stacked generalization or stacking. The key idea of this approach is to combine multiple classifiers by training the higher-level classifier by the output predictions of the lower-level classifiers, estimated via cross validation as follow:

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

The dataset D consists of a set of feature vectors $D = < x_i, y_i >$, where the vector x_i is a system call feature vector of an application and the type y_i indicates whether this application is benign or malicious. For j-cross validation, the given dataset D is divided into j almost equal-sized disjoint folds: $d_1, d_2, \ldots, d_j$. The lower-level classifiers $C_1, C_2, \ldots, C_n$ are generated by training the learning algorithms $L_1, L_2, \ldots, L_n$ respectively on $j-1$ folds and using the jth fold for testing the classifiers. The predictions of the lower-level classifiers on a feature vector x_k and its original class type y_k, form a new feature vector for the upper-level classifier as illustrated in Figure 5.6. This process is repeated with each fold. At the end of this process, the union of the prediction feature vectors represent the new dataset that is used for training upper-level learning algorithm L_M and producing meta-level classifier C_M.

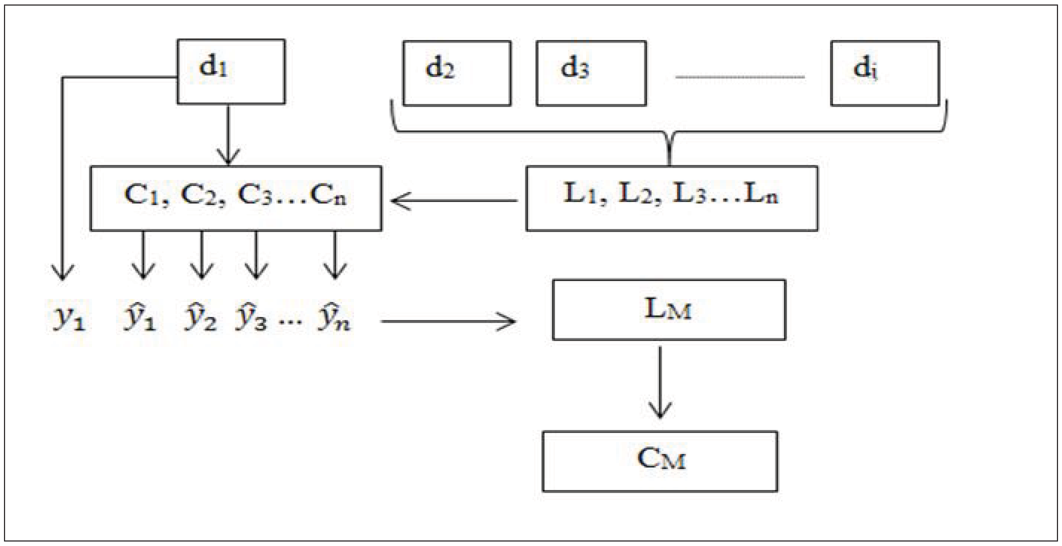


Figure 5.6 Stacking training phase

In order to classify a new instance, it is first applied to lower-level classifiers. The predictions of these classifiers are concatenated with the original class type and form the upper-level feature vector. The upper-level classifier C_M assigns class type to this vector, which is the final class type prediction of the new instance. The classification process is illustrated in Figure 5.7.

The main advantage of the stacking approach is its simplicity. It is easy to treat the classifiers as black boxes, without considering the details of their implementations. As well as using the

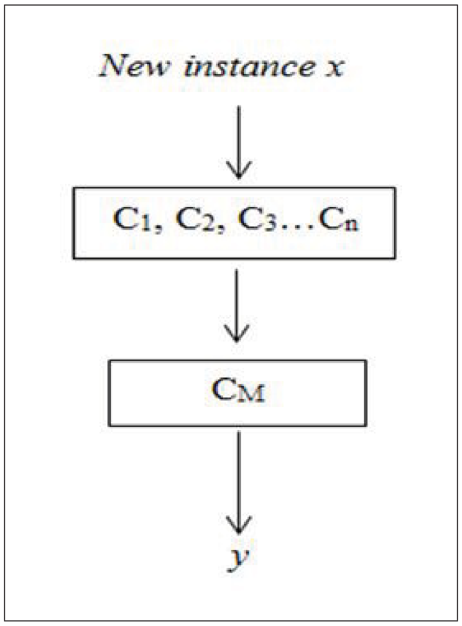


Figure 5.7 Stacking classification phase

predictions of lower-level classifiers as inputs for the upper-level allows to correct the misclassified instances and improve the overall prediction accuracy.

5.2.1 Experiment results

The experimental results follows the same approach as one used in chapters 4 and 5. In order to evaluate the performance of hybrid classifiers and compared with single classifiers, system call traces of benign and malicious applications are collected. The benign applications represent the hundred most downloaded free applications. These applications are downloaded from Android's official market. The malicious applications are hundred real malwares that are downloaded from (blog Contagio: Mobile malware, 2012). All applications are installed and executed on Android 2.3 version. The adopted feature vector is bag of system calls as presented in chapter 3, where each system call is presented by its frequency.

The used metrics to evaluated the accuracy performance of the classifiers are: the *false positive* and the overall *accuracy*. The used machine learning classifiers in this chapter are obtained from *weka* machine learning visual package (university of waikato, 2012). Due to the limitation of resources of smartphone, in this experiment, we evaluate the simplest hybrid classifier of two levels and only one classifiers in each level. The selected classifiers are: support vector machine (SVM), artificial neural network (ANN), naive bayes (NB) and logistic regression (LR).

Firstly, the classifiers (single and hybrid) are evaluated using the raw traces. The prediction accuracy of SVM, ANN and hybrid SVM-ANN classifiers is reported in Table 5.7. SVM has better performance than ANN classifier. However, Hybrid SVM-ANN classifier performs better than both single classifiers. It significantly reduces the false positive and improves the prediction accuracy.

Table 5.7 SVM, ANN and hybrid SVM-ANN classifiers accuracy performance.

Classifier	**Accuracy**	**False Positive**
SVM	$92,50\%$	$8,50\%$
ANN	$89,60\%$	$11,50\%$
Hybrid (SVM-ANN)	96%	4%

Table 5.8 shows the prediction accuracy of 3 classifiers, naïve bayes (NB), logistic regression (LR) and the corresponding hybrid NB-LR. Logistic regression is better than naïve bayes. However, hybrid NB-LR classifier performs better than Logistic regression classifier.

Table 5.8 NB, LR and Hybrid NB-LR classifiers accuracy performance.

Classifier	**Accuracy**	**False Positive**
Naïve bayes	$91,30\%$	9%
Logistic regression	$92,30\%$	8%
Hybrid (NB-LR)	$94,10\%$	6%

Table 5.9 shows the high prediction accuracy of hybrid classifier SVM-NB relative to their single classifiers SVM and Naïve Bayes. SVM has lower FP rate (8.50%) and higher accuracy

(92.50%) than Naïve Bayes. However, hybrid classifier SVM-NB reduces false positive and increases accuracy.

Table 5.9 SVM, NB and Hybrid SVM-NB classifiers performance.

Classifier	Accuracy	False Positive
SVM	$92,50\%$	$8,50\%$
Naïve bayes	$91,30\%$	9%
Hybrid (SVM-NB)	$94,50\%$	$5,50\%$

Table 5.10 illustrates the efficient performance of a new possible hybrid classifier LR-ANN relative to their individual classifier logistic regression (LR) and ANN. Logistic regression is more efficient than ANN. Hybrid classifier LR-ANN reduces FP rate and improves the accuracy.

Table 5.10 LR, ANN and Hybrid LR-ANN classifiers accuracy performance.

Classifier	Accuracy	False Positive
Logistic regression	$92,30\%$	8%
ANN	$89,60\%$	$11,50\%$
Hybrid (LR-ANN)	96%	4%

Summarizing the results of the above tables, we can state that the individual machine learning classifiers have good performance on smartphone system calls dataset. The best classifier was SVM with 8% false positive and 92.50% accuracy. The worst classifier was ANN with 11.50% false positive and 89.60% accuracy. The hybrid classifiers achieved better performance than individual ones. They have the lowest false positive (4%) and the highest accuracy (96%). The worst prediction accuracy of hybrid classifier is LR-NB classifier with 6% of false positive and 94.10% of accuracy. This prediction accuracy is better than the best individual classifiers (SVM with 8% of false positive and 92.50% of accuracy). The hybrid classifiers improves the classification accuracy compared to single classifiers.

5.3 Discussion and conclusion

This chapter has two main objectives: (i) investigate the impact of system call filtering and abstraction process on the performance of machine learning classifiers, (ii) investigate different models of benign behavior by implementing two machine learning classifiers from different categories.

Comparing Table 5.2 and Table 5.4, it is clear that the capabilities of SVM classifier to discriminate between malicious and benign applications are better than the capabilities of the *k*-means classifier. For example, for the raw traces, the SVM classifier detects around 92.30% of malware and misses only around 7.66%, while the one-class *k*-means detects around 81.80% of malware and misses around 18.20%. The significant difference between classifiers in the five performance metrics clearly illustrates that the two-class SVM classifier is more discriminative and suitable to malware detection application. The SVM classifier is a discriminative classifier, and its training process is based on learning the boundaries of the classes, while *k*-means is generative classifier, and its training is based on learning the density distribution of the target class.

Unlike the accuracy performance, there are not huge differences between the space complexity of the two selected classifiers. However, the time complexity of the SVM classifier is better than *k*-means classifier. Table 5.11 summarizes the time and space complexities of the training phase of the two classifiers.

The testing phase complexity is usually very fast since the testing phase uses the learnt model for classification.

Table 5.11 SVM and *k*-means time and memory complexities.

Classifier	**Time complexity**	**Space complexity**
SVM	$O(n^2)$ or $O(n^3)$	$O(n)$
k-means	$O(n \times d \times k \times l)$	$O((n+k) \times d)$

As illustrated previously, the filtering and abstraction process has positive impacts on the accuracy performance of the two classifiers. For example, the false positive of the *k*-means is improved by reducing the false alarm rate from 22,50% to 8,90% as well as the SVM false positive is reduced from 7,70% to 0%.

The filtering and abstraction process also impacts the time and space complexities positively. Reducing the number of features d, the number of centroid k and the parameter C impacts positively the space and time complexities of the *k*-means and the SVM as illustrated in table 5.11. The number of features d is reduced by more than 50%, which means the filtering and abstraction process reduces the memory usage by more than 50%.

In conclusion, the SVM is more accurate than *k*-means. Therefore, it is preferable for anomaly-based detection system when a sufficient number of malicious and benign applications are available. The *k*-means is significantly improved by adding the filtering and abstraction process, and it can be used in anomaly-based detection system when only one class dataset is available.

Further study and evaluation would have to be done eventually with other classifiers such as one-class SVM and GMM for one-class classifiers, as well as decision tree and naïve bayes, etc. for two-class classifiers with system calls dataset and other dataset, such as requested permission and API.

Hybrid machine learning classifiers improve the prediction precision of individual classifiers with extra space and time complexities. Filtering and abstraction process improves the prediction precision of the SVM (individual classifiers) more than hybrid classifiers and with less space and time complexities.

CONCLUSION

In this Thesis, we address the malware detection problem on smartphone systems. Smartphone popularity has grown over the last few years due to different reasons, such as:

- Smartphone system's interface is attractive, simple and easy to use. As a result, new users get acquainted with the use of different functions quickly.
- Smartphone systems allow users to stay connected with personal and business life. Users have Internet access all the time, and they can send emails, update Facebook profiles, access bank account information, pay bills, and accept or decline appointments. These facts allow users to save time.
- Nowadays, smartphone systems are affordable. Smartphone companies keep improving the smartphone systems capabilities and functions, while at the same time lowering prices.
- The most important reason of this popularity is the third-party applications. Users can develop and run applications on their devices. These applications add more functions that make smartphone systems so desirable, useful and serviceable.

Due to the functions, services and popularity of smartphone systems, users store confidential information and make confidential operations and transactions on their devices. The popularity of smartphone systems along with their ability to store confidential information are two main factors that attract cyber criminals and malware developers.

Smartphone malwares are becoming more and more dangerous and complex to detect. The last report of kaspersky lab (Chebyshev and Unuchek, 2014) reported shocking results. Over 143,000 new malwares were detected, around 4 million installation packages were used to distribute malwares, and approximately 10 million unique malicious installation packages were detected. These results show the rapid increase and the serious threats of smartphone malwares.

In light of this rapid increase and advanced techniques used by malwares targeting smartphone systems, there is crucial need to develop effective solutions to protect these systems. The

Rapport-Gratuit.com

literature review on this subject has shown that the two main techniques used to detect malwares were: signature-based techniques and anomaly-based techniques (Amamra *et al.*, 2012b). Each technique has its strengths and drawbacks, and no single technique is expected to detect all types of malwares. Therefore, they are complementary in detecting malwares.

In this Thesis, we focus on anomaly-based detection techniques. Anomaly-based techniques are more resisting to evasion attacks and obfuscation techniques, as well as they can detect unknown malwares and variant of known malwares. However, these techniques still suffer from two main problems:

- They still need more investigations and improvements to increase the prediction accuracy. The main factors impacting the prediction accuracy are:
  - The data used to represent the application behavior is not representative enough and/or their amount is insufficient.
  - The behavior model describing the benign application is inadequate and its parameters are poorly optimized.
- Smartphone systems have limited resources environments which limit the complexity of the adopted anomaly detection techniques. Thereby, the solution should consume as less system resources as possible.

The objective of this Thesis is to propose novel and efficient anomaly detection solutions to detect malwares on smartphone system based Android OS. The target solution should be accurate, adaptable and scalable. In addition, it should be compacted and optimised enough to respect the resource limitations of the smartphone systems.

To reach this objective, this Thesis introduced contributions on three different levels: (1)literature review study, (2) the data used to represent the behavior of applications and (3) the used benign behavior model. The contributions are presented briefly in the next paragraphs.

To understand the current trends of smartphone malware detection techniques and identify the possible paths to improve the existing solutions or propose new ones, a survey of the malware detection techniques for smartphone systems is introduced in Chapter 2. This survey presents a comprehensive review of the existing detection techniques and provides a taxonomy of these techniques according to well defined rules. These rules are: the reference behavior, the analysis technique, the benign behavior model and the used data to represent the benign behavior (Amamra *et al.*, 2012b).

Chapter 3 presents dataset level contributions, which provide representative dataset that can be used to design accurate anomaly detection solutions. In this Thesis, The Linux system call trace is selected. System call traces have several advantages. they are simple, closer to the real behavior of applications and good discriminator. The dataset contribution is twofold: the feature vector representation (Amamra *et al.*, 2013) and the system call filtering and abstraction process (Amamra *et al.*, 2014).

Feature vector representation is the format adopted to organize and encode the data. In prior work, successive system calls and frequency distribution (bag) of system calls feature vectors are used in the prior work. A new one, the frequency of short patterns system calls is examined, this feature vector combines successive and frequency information of system calls. The experiment results illustrate the significant impact of the feature vector representation on the prediction accuracy of the classifiers. The pattern frequency feature vectors improve the prediction accuracy of all tested classifiers. For example, SVM classifier has low performance with the bag of system calls representation (92.50% accuracy rate and 8.50% false positive rate). This performance is enhanced by using (two and three)-pattern frequency representation to 100% accuracy rate and 0% false positive rate on limited number of applications.

Pattern frequency system calls feature vectors have extra computational and space cost. The memory complexity of the l-pattern frequency of system calls representation is in $O(N \times |\sum|^l)$, where N is the number of traces, $|\sum|$ is number of system calls and l is length of pattern. It

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

consumes more memory than the bag of system calls representation and less than the successive system calls representation.

The second contribution of dataset level is the process for filtering and abstracting system calls. The is in fact the one of the two main contributions of this Thesis. This process consists of two phases: the filtering phase and the abstraction phase (Amamra *et al.*, 2014). The first one removes the system calls that do not represent the main behavior of an application, such as memory management system calls, system calls to check the available resources, system calls to obtain process information, inter-process system calls and failed system calls. The abstraction module considers the system calls of similar functions and different names as one system call. For example, read(), readv(), pread() and fread() are unified to read() system calls. The filtering and abstraction process produces refined traces that are much more compact than raw traces. This improves the system resources consumption and it is much more representative of the main behavior of applications, which positively impacts the prediction accuracy of the solution. The evaluation of the refined traces vs raw traces is presented in Chapter 4 and Chapter 5.

Chapter 4 presents a novel canonical database normal behavior model. This model is an extension to the classical lightweight anomaly detection approach proposed by (Forrest *et al.*, 1996). The normal behavior of a legitimate privileged process is represented by a database composed of all the unique system call patterns of a given length *k* encountered during a training phase. In our context, there are millions of Android applications and building the specific database for each of these applications. Therefore, this approach has the following limitations:

- It is impossible for third party to build separated databases for all these applications.
- It is not possible to detect a malware for which no legitimate database is available.
- Any application update will increase false positive.

Hence, our solution builds a canonical database representing generic benign behavior of Android applications. This database is constructed from a limited number of representative applications. Once the canonical database has been built, it can be used to detect anomaly of new applications. When a pattern does not exist in the database, such pattern is considered as a mismatch and may present an anomalous behavior (Amamra *et al.*, 2014). The efficiency of this approach is based on two things:

- The algorithm used to check the existence of the new patterns in the database.
- The method used to evaluate the anomaly strength of a new application.

Due to their efficiency, multi-pattern matching algorithms are suitable to verify the existence of new patterns in the canonical database. In (Amamra *et al.*, 2012a), we have selected four algorithms using different matching approaches. These algorithms are: Aho-Corasick (AC) algorithm (Aho and Corasick, 1975; Amamra *et al.*, 2012a) uses finite automaton approach, Commentz-Walter (CW) algorithm (Commentz-Walter, 1979; Amamra *et al.*, 2012a) uses finite automaton heuristic approach, Wu-Manber (WM) algorithm (Wu *et al.*, 1994; Amamra *et al.*, 2012a) uses hash-heuristic approach and Set backward Oracle Matching (SBOM) algorithm (Amamra *et al.*, 2012a) uses factor approach. Those algorithms have been implemented and tested on a smartphone device. We identify the used memory and the available one, we then determine the budget of each algorithm according to the number of patterns. The AC algorithm has been selected because it runs in linear time in all cases, which means that the algorithm has stable performance.

In prior work, the anomaly strength of an application trace is measured by two methods: the number of mismatches without considering the level of mismatches (Forrest *et al.*, 1996, 2008) and the maximum level of mismatches without considering the number of mismatches (Hofmeyr *et al.*, 1998). These two methods are not sufficient representative of the anomaly of a trace (as discussed in section 4.2). Therefore, a new method called weighted anomaly level is introduced to measure the anomaly (Amamra *et al.*, 2014). This method takes in consideration both the number of mismatches and the maximum level of mismatches. The experiments show this method is very representative of the anomaly of applications traces.

Canonical database approach is evaluated on both raw and refined traces and measuring the anomaly by maximum level of mismatches and weighted anomaly level. The experiment results show the approach on the raw traces has good performance, where it consumes limited system resources (memory and CPU) and has good prediction accuracy. On refined traces, the accuracy and execution performances are improved, i.e. the system resources consumption is lesser and the prediction accuracy is higher. This is the second main contribution of this Thesis. This approach can be seen as the last line of defence in a defence-in-depth mechanism. Ideally, a pattern database should be provided for an application by the application developer. Such database should be very accurate.

Chapter 5 reviews and compares the efficiency of two main categories of machine learning classifiers and evaluate the impact of filtering and abstraction process on each category (Amamra *et al.*, 2015). Machine learning classifiers can be generative classifiers (one-class classifier) or discriminative classifiers(multi-class classifier) (Ulusoy and Bishop, 2006; Bishop *et al.*, 1995). SVM classifier represents the discriminative classifier category and *k*-means classifier represents the generative classifier category. The experiment results show the prediction accuracy performances of the two classifiers, and it is clear the two-class SVM classifier is more discriminative and suitable for malware detection applications. The filtering and abstraction process positively impacts both the resource usage and prediction accuracy of the two classifiers. The false positive rate is very important measure to evaluate the efficiency of anomaly detection solutions. We compare the false positive rate of the two classifiers before and after filtering abstraction process. The *k*-means false alarm rate can be reduced from 22.50% to 8.90% after the filtering and abstraction process. The SVM false positive rate can be reduced from 7.70% to 0%.

Chapter 6 presents hybrid machine learning classifiers. Hybrid classifier is a set of classifiers whose single predictions are combined in some way to classify new samples (Amamra *et al.*, 2012c). Hybrid classifiers use the output of the classifiers without considering details of their implementation.

The study presented in this Thesis can be extended and enhanced in future work in three main directions: (1) more representative dataset of application behavior, (2) more discriminative algorithms between benign and malware behavior, and (3) hybrid framework use system calls to represent benign behavior and others to represent malwares.

In first direction, system calls filtering and abstraction process still needs more investigation and improvement. Fine-grained filtering criteria should produce more representative data of application behavior. System calls data can be more representative of application behavior by adding addition information about application behavior such as arguments of system calls and resource access events. Explore other data than system calls especially static ones could be more representative and discriminative.

In the second direction, new algorithms and models should be investigated, explored and adapted with smartphone system anomaly detection specifications, such as new machine learning classifier never investigated, finite state machine (FSM) to model the normal behavior. In this model, the states describe the past, the transitions indicate the change from state to other according to specific conditions and the actions describe the activities. FSM model have shown good performance in network anomaly detection.

In the last direction, hybrid framework consists of two databases, one represents the generic normal behavior and other represents malwares generic behavior. This framework should enhance the detection rate and reduces the false positive rate.

The contributions of this Thesis have been the subject of four refereed international conference papers and two journal papers as follows:

- International conference papers
  - Amamra, Abdelfattah, Chamseddine Talhi, and Jean-Marc Robert. 2012. "Performance Evaluation of Multi-pattern Matching Algorithms on Smartphone". In the Seventh International Conference on Broadband, Wireless Computing, Communication and Applications. *Published*

- Amamra, Abdelfattah, Chamseddine Talhi, and Jean-Marc Robert. 2012. "Smartphone malware detection: From a survey towards taxonomy". In 7th International Conference on Malicious and Unwanted Software (MALWARE). *Published*
- Amamra, Abdelfattah, Chamseddine Talhi, Jean-Marc Robert, and Martin Hamiche. 2012. "Enhancing Smartphone Malware Detection Performance by Applying Machine Learning Hybrid Classifiers". In Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity. *Published*
- Amamra, Abdelfattah, Chamseddine Talhi, and Jean-Marc Robert. 2013. Impact of dataset representation on smartphone malware detection performance. International IFIP Conference on Trust Management VII. *Published*

- Journal papers
  - Amamra, Abdelfattah, Jean-Marc Robert, and Chamseddine Talhi. 2015. "Enhancing malware detection for Android systems using a system call filtering and abstraction process". Security and Communication Networks. *Published*
  - Amamra, Abdelfattah, Jean-Marc Robert, Andrien Abraham and Chamseddine Talhi. 2014. "Generative vs Discriminative Classifiers for Android Anomaly-based Malware Detection System using System Calls Filtering and Abstraction Process". Security and Communication Networks. *Submitted*

BIBLIOGRAPHY

Aafer, Yousra, Wenliang Du, and Heng Yin. 2013. "DroidAPIMiner: Mining API-level features for robust malware detection in android". In *Security and Privacy in Communication Networks*. p. 86–103. Springer.

Aho, Alfred V and Margaret J Corasick. 1975. "Efficient string matching: an aid to bibliographic search". *Communications of the ACM*, vol. 18, n° 6, p. 333–340.

Allauzen, Cyril, Maxime Crochemore, and Mathieu Raffinot. 2001. "Efficient Experimental String Matching by Weak Factor Recognition*". In *Combinatorial Pattern Matching Conference*. p. 51–72. Springer.

Alpaydin, Ethem, 2004. *Introduction to machine learning*.

Amamra, Abdelfattah, Chamseddine Talhi, and Jean-Marc Robert. 2012a. "Performance Evaluation of Multi-pattern Matching Algorithms on Smartphone". In *the Seventh International Conference on Broadband, Wireless Computing, Communication and Applications*. p. 329–334. IEEE Computer Society.

Amamra, Abdelfattah, Chamseddine Talhi, and Jean-Marc Robert. 2012b. "Smartphone malware detection: From a survey towards taxonomy". In *7th International Conference on Malicious and Unwanted Software (MALWARE)*. p. 79–86. IEEE.

Amamra, Abdelfattah, Chamseddine Talhi, Jean-Marc Robert, and Martin Hamiche. 2012c. "Enhancing Smartphone Malware Detection Performance by Applying Machine Learning Hybrid Classifiers". In *Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity*. (Jeju Island, Korea 2012), p. 131–137. Springer.

Amamra, Abdelfattah, Chamseddine Talhi, and Jean-Marc Robert. 2013. Impact of dataset representation on smartphone malware detection performance. *International IFIP Conference on Trust Management VII*, p. 166–176. Springer.

Amamra, Abdelfattah, Jean-Marc Robert, and Chamseddine Talhi. 2014. "Enhancing malware detection for Android systems using a system call filtering and abstraction process". *Security and Communication Networks*.

Amamra, Abdelfattah, Andrien Abraham Robert, Jean-Marc, and Chamseddine Talhi. 2015. "Generative vs Discriminative Classifiers for Android Anomaly-Based Detection System using System Calls Filtering and Abstraction Process". *Submited to Security and Communication Networks Journal*.

Amor, Nahla Ben, Salem Benferhat, and Zied Elouedi. 2004. "Naive bayes vs decision trees in intrusion detection systems". In *the ACM symposium on Applied computing*. p. 420–424. ACM.

Arp, Daniel, Michael Spreitzenbarth, Malte Hübner, Hugo Gascon, and Konrad Rieck. 2014. "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket". In *Network and Distributed System Security Symposium (NDSS)*. p. 300–305.

Arzt, Steven, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps". In *the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. p. 29. ACM.

Aung, Zarni and Win Zaw. 2013. "Permission-based Android malware detection". *International Journal Of Scientific & Technology Research*, vol. 2, n° 3.

Bartel, Alexandre, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2014. "Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges And Solutions for Analyzing Android". *IEEE Transactions on Software Engineering*, vol. 40, n° 6, p. 617- 632.

Bishop, Christopher M et al., 1995. *Neural networks for pattern recognition*.

Blasing, Thomas, Leonid Batyuk, A-D Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. 2010. "An android application sandbox system for suspicious software detection". In *5th international conference on Malicious and unwanted software (MALWARE)*. p. 55–62. IEEE.

blog Contagio: Mobile malware. 2012. "Mobile Malwares". <http://contagiodump.blogspot.ca/>.

Bordes, Antoine, Seyda Ertekin, Jason Weston, and Léon Bottou. 2005. "Fast kernel classifiers with online and active learning". *The Journal of Machine Learning Research*, vol. 6, p. 1579–1619.

Bose, Abhijit, Xin Hu, Kang G Shin, and Taejoon Park. 2008. "Behavioral detection of malware on mobile handsets". In *the 6th international conference on Mobile systems, applications, and services*. p. 225–238. ACM.

Boyer, Robert S and J Strother Moore. 1977. "A fast string searching algorithm". *Communications of the ACM*, vol. 20, n° 10, p. 762–772.

Buennemeyer, Timothy K, Theresa M Nelson, Lee M Clagett, John Paul Dunning, Randy C Marchany, and Joseph G Tront. 2008. "Mobile device profiling and intrusion detection using smart batteries". In *the 41st AnnualInternational Conference on System Sciences*. p. 296–296. IEEE.

Burguera, Iker, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. "Crowdroid: behavior-based malware detection system for android". In *the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. p. 15–26. ACM.

Cha, Sang Kil, Iulian Moraru, Jiyong Jang, John Truelove, David Brumley, and David G Andersen. 2011. "SplitScreen: Enabling efficient, distributed malware detection". *Communications and Networks, Journal of*, vol. 13, n° 2, p. 187–200.

Chandola, Varun, Arindam Banerjee, and Vipin Kumar. 2009. "Anomaly detection: A survey". *ACM Computing Surveys (CSUR)*, vol. 41, n° 3, p. 15.

Chang, Chih-Chung and Chih-Jen Lin. 2011. "LIBSVM: A Library for Support Vector Machines". *ACM Trans. Intell. Syst. Technol.*, vol. 2, n° 3, p. 27:1–27:27.

Chapelle, Olivier, Vladimir Vapnik, Olivier Bousquet, and Sayan Mukherjee. 2002. "Choosing multiple parameters for support vector machines". *Machine learning*, vol. 46, n° 1-3, p. 131–159.

Chebyshev, Victor and Roman Unuchek. 2014. "Mobile Malware Evolution: 2013". http://securelist.com/analysis/kaspersky-security-bulletin/58335/mobile-malware-evolution-2013/. Accessed: 15 June 2014.

Chen, Thomas M and Cyrus Peikari. 2008. "Malicious software in mobile devices". *Handbook of Research on Wireless Security*, vol. 1, p. 1–10.

Chiang, MarkMing-Tso and Boris Mirkin. 2010. "Intelligent Choice of the Number of Clusters in *k*-Means Clustering: An Experimental Study with Different Cluster Spreads". *Journal of Classification*, vol. 27, n° 1, p. 3-40.

Chin, Erika, Adrienne Porter Felt, Kate Greenwood, and David Wagner. 2011. "Analyzing inter-application communication in Android". In *the 9th international conference on Mobile systems, applications, and services*. p. 239–252. ACM.

Christodorescu, Mihai and Somesh Jha. 2006. *Static analysis of executables to detect malicious patterns*. Technical report.

cisco security. 2011. "Mobile Malware Evolution: An Overview, Part 4". http://www.cisco.com/web/about/security/intelligence/virus-worm-diffs.html. Accessed: 02 June 2014.

CNCCS. 2010. "Smartphone Malware". http://www.a51.nl/storage/pdf/CNCCS_Smartphone_Malware_Full_Report_Translated_06_7_11_FINAL.pdf. Accessed: 03 June 2014.

Commentz-Walter, Beate, 1979. *A string matching algorithm fast on the average*.

Cortes, Corinna and Vladimir Vapnik. 1995. "Support-vector networks". *Machine learning*, vol. 20, n° 3, p. 273–297.

Dai, Shuaifu, Yaxin Liu, Tielei Wang, Tao Wei, and Wei Zou. 2010. "Behavior-based malware detection on mobile phone". In *6th International Conference on Wireless Communications Networking and Mobile Computing (WiCOM)*. p. 1–4. IEEE.

Rapport-Gratuit.com

developer.android.com. 2014a. "System Permissions". http://developer.android.com/guide/topics/security/permissions.html. Accessed: 02 August 2014.

developer.android.com. 2014b. "Application Fundamentals". http://developer.android.com/guide/components/fundamentals.html. Accessed: 02 June 2014.

Dietterich, Thomas G. 2000. "An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization". *Machine learning*, vol. 40, n° 2, p. 139–157.

Dunham, Ken, 2008. *Mobile malware attacks and defense*.

Džeroski, Saso and Bernard Ženko. 2004. "Is combining classifiers with stacking better than selecting the best one?". *Machine learning*, vol. 54, n° 3, p. 255–273.

Enck, William, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. 2011. "A Study of Android Application Security.". In *USENIX security symposium*. p. 2.

Enck, William, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. 2014. "TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones". *Communications of the ACM*, vol. 57, n° 3, p. 99–106.

Engineers' Forum, Magazine. 2013. "The Evolution of the Mobile Phone". http://www.ef.org.vt.edu/2013/11/the-evolution-of-the-mobile-phone.html. Accessed: 01 August 2014.

F-Secure. 2013. "MOBILE THREAT REPORT". http://www.f-secure.com/documents/996508/1030743/Mobile_Threat_Report_Q3_2013.pdf. Accessed: 02 June 2014.

Fawcett, Tom. 2006. "An introduction to ROC analysis". *Pattern recognition letters*, vol. 27, n° 8, p. 861–874.

Felt, Adrienne Porter, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. 2011. "Android permissions demystified". In *Proceedings of the 18th ACM conference on Computer and communications security*. p. 627–638. ACM.

Forrest, Stephanie, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. 1996. "A sense of self for unix processes". In *IEEE Symposium on Security and Privacy*. p. 120–128. IEEE.

Forrest, Stephanie, Steven Hofmeyr, and Anil Somayaji. 2008. "The evolution of system-call monitoring". In *Annual Computer Security Applications Conference*. p. 418–430. IEEE.

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

Gandhewar, Nisarg and Rahila Sheikh. 2010. "Google Android: An emerging software platform for mobile devices". *International Journal on Computer Science and Engineering*, vol. 1, n° 1, p. 12–17.

Giacinto, Giorgio, Roberto Perdisci, and Fabio Roli. 2005. Network intrusion detection by combining one-class classifiers. *Image Analysis and Processing–ICIAP 2005*, p. 58–65. Springer.

Gibler, Clint, Jonathan Crussell, Jeremy Erickson, and Hao Chen, 2012. *AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale*.

Gong, Yihong. 2008. Video content analysis using machine learning tools. *Encyclopedia of Multimedia*, p. 905–911. Springer.

Grace, Michael C, Yajin Zhou, Zhi Wang, and Xuxian Jiang. 2012. "Systematic Detection of Capability Leaks in Stock Android Smartphones.". In *Network and Distributed System Security Symposium*.

Guojun, PENG, SHAO Yuru, WANG Taige, ZHAN Xian, and ZHANG Huanguo. 2012. "Research on Android Malware Detection and Interception Based on Behavior Monitoring". *Wuhan University Journal Of natural Sciences*, vol. 17, n° 5, p. 421-427.

Hodge, Victoria J and Jim Austin. 2004. "A survey of outlier detection methodologies". *Artificial Intelligence Review*, vol. 22, n° 2, p. 85–126.

Hofmeyr, Steven A, Stephanie Forrest, and Anil Somayaji. 1998. "Intrusion detection using sequences of system calls". *Journal of computer security*, vol. 6, n° 3, p. 151–180.

Hsieh, Cheng-Kang, Hossein Falaki, Nithya Ramanathan, Hongsuda Tangmunarunkit, and Deborah Estrin. 2013. "Performance evaluation of android IPC for continuous sensing applications". *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 16, n° 4, p. 6–7.

Hsu, Chih-Wei, Chih-Chung Chang, Chih-Jen Lin, et al. 2003. "A practical guide to support vector classification". https://www.cs.sfu.ca/people/Faculty/teaching/726/spring11/svmguide.pdf.

IDC. 2011. "Worldwide Smartphone Market Expected to Grow 55Approach Shipments of One Billion in 2015, According to IDC". http://www.businesswire.com/news/home/20110609005403/en/Worldwide-Smartphone-Market-Expected-Grow-55-2011#.VGI9z_mG98E. Accessed: 06 June 2014.

IDC. 2013. "Tablet Shipments Forecast to Top Total PC Shipments in the Fourth Quarter of 2013 and Annually by 2015, According to IDC". http://www.idc.com/getdoc.jsp?containerId=prUS24314413. Accessed: 01 August 2014.

IDC. 2014. "Smartphone OS Market Share, Q2 2014". http://www.idc.com/prodserv/smartphone-os-market-share.jsp. Accessed: 15 August 2014.

Rapport-gratuit.com
LE NUMERO 1 MONDIAL DU MÉMOIRES

Idika, Nwokedi and Aditya P Mathur. 2007. "A survey of malware detection techniques". *Purdue University*, vol. 48.

Inoue, Hajime and Anil Somayaji. 2007. "Lookahead pairs and full sequences: a tale of two anomaly detection methods". In *In Proc. of the 2nd Annual Symposium on Information Assurance*. p. 9–19.

Jacob, Grégoire, Hervé Debar, and Eric Filiol. 2008. "Behavioral detection of malware: from a survey towards an established taxonomy". *Journal in computer Virology*, vol. 4, n° 3, p. 251–266.

Jain, Anil K. 2010. "Data clustering: 50 years beyond K-means". *Pattern Recognition Letters*, vol. 31, n° 8, p. 651 - 666.

Jiang, Xuxian. 2011. "Security Alert: New NickiBot Spyware Found in Alternative Android Markets". http://www.csc.ncsu.edu/faculty/jiang/NickiBot/. Accessed: 02 June 2014.

John, Jackson. 2013. "Worldwide and U.S. Mobile Applications Download and Revenue 2013–2017 Forecast: The App as the Emerging Face of the Internet". http://www.idc.com/getdoc.jsp?containerId=241295. Accessed: 01 August 2014.

Kabiri, Peyman and Ali A Ghorbani. 2005. "Research on Intrusion Detection and Response: A Survey.". *IJ Network Security*, vol. 1, n° 2, p. 84–102.

Kanungo, Tapas, David M Mount, Nathan S Netanyahu, Christine D Piatko, Ruth Silverman, and Angela Y Wu. 2002. "An efficient k-means clustering algorithm: Analysis and implementation". *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 24, n° 7, p. 881–892.

Kelly, James. 2006. "An Examination of Pattern Matching Algorithms for Intrusion Detection Systems". Master's thesis, Carleton University.

Kim, Hahnsang, Joshua Smith, and Kang G Shin. 2008. "Detecting energy-greedy anomalies and mobile malware variants". In *the 6th international conference on Mobile systems, applications, and services*. p. 239–252. ACM.

Kim, Seil, Jae Ik Cho, Hee Won Myeong, and Dong Hoon Lee. 2012. "A study on static analysis model of mobile application for privacy protection". In *Computer Science and Convergence*. p. 529–540. Springer.

Kohavi, Ron et al. 1995. "A study of cross-validation and bootstrap for accuracy estimation and model selection". *IJCAI*, vol. 14, n° 2, p. 1137–1145.

Kouzinopoulos, Charalampos S and Konstantinos G Margaritis. 2011. "A performance evaluation of the preprocessing phase of multiple keyword matching algorithms". In *The 15th Panhellenic Conference on Informatics (PCI)*. p. 85–89. IEEE.

La Polla, Mariantonietta, Fabio Martinelli, and Daniele Sgandurra. 2013. "A survey on security for mobile devices". *Communications Surveys & Tutorials, IEEE*, vol. 15, n° 1, p. 446–471.

Lin, Ying-Dar, Yuan-Cheng Lai, Chien-Hung Chen, and Hao-Chuan Tsai. 2013. "Identifying android malicious repackaged applications by thread-grained system call sequences". *Computers & Security*, vol. 39, p. 340–350.

Lokhande, Bhushan and Sunita Dhavale. 2014. "Overview of information flow tracking techniques based on taint analysis for Android". In *International Conference on Computing for Sustainable Global Development (INDIACom)*. p. 749–753. IEEE.

lookout mobile security report. 2011. "Mobile Threat Report 2011". https://www.lookout.com/resources/reports/mobile-threat-report-2011. Accessed: 02 June 2014.

Maslennikov, Denis. 2011. "Mobile Malware Evolution: An Overview, Part 4". http://securelist.com/large-slider/36350/mobile-malware-evolution-an-overview-part-4/. Accessed: 02 June 2014.

Mukkamala, Srinivas, Guadalupe Janoski, and Andrew Sung. 2002. "Intrusion detection using neural networks and support vector machines". In *the International Joint Conference onNeural Networks (IJCNN)*. p. 1702–1707. IEEE.

Mulliner, Collin and Charlie Miller. 2009. "Injecting SMS messages into smart phones for security analysis". In *USENIX Workshop on Offensive Technologies (WOOT)*.

Mulliner, Collin and Giovanni Vigna. 2006. "Vulnerability analysis of mms user agents". In *22nd Annual Conference on Computer Security Applications*. p. 77–88. IEEE.

Newsome, James and Dawn Song. 2005. *Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software*. Technical report.

Passeri, Paolo. 2011. "One Year Of Android Malware (Full List)". http://hackmageddon.com/2011/08/11/one-year-of-android-malware-full-list/. Accessed: 05 June 2014.

Peiravian, Naser and Xingquan Zhu. 2013. "Machine Learning for Android Malware Detection Using Permission and API Calls". In *IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI)*. p. 300–305. IEEE.

Platt, John et al. 1999. "Fast training of support vector machines using sequential minimal optimization". *Advances in kernel methods—support vector learning*, vol. 3.

Rastogi, Vaibhav, Yan Chen, and Xuxian Jiang. 2013. "Droidchameleon: evaluating android anti-malware against transformation attacks". In *8th ACM SIGSAC symposium on Information, computer and communications security*. p. 329–334. ACM.

Ratle, Frédéric, Mikhail Kanevski, Anne-Laure Terrettaz-Zufferey, Pierre Esseiva, and Olivier Ribaux. 2007. A comparison of one-class classifiers for novelty detection in forensic case data. *Intelligent Data Engineering and Automated Learning-IDEAL 2007*, p. 67–76. Springer.

Sahs, Justin and Latifur Khan. 2012. "A machine learning approach to Android malware detection". In *European Intelligence and Security Informatics Conference (EISIC)*. p. 141–147. IEEE.

Sanz, Borja, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Javier Nieves, Pablo G Bringas, and Gonzalo Álvarez Marañón. 2013. "MAMA: Manifest Analysis for Malware Detection in Android". *Cybernetics and Systems*, vol. 44, n° 6-7, p. 469–488.

Schmidt, Aubrey-Derrick. 2011. "Detection of Smartphone Malware.". PhD thesis, Berlin Institute of Technology.

Schmidt, Aubrey-Derrick, Frank Peters, Florian Lamour, Christian Scheel, Seyit Ahmet Çamtepe, and Şahin Albayrak. 2009. "Monitoring smartphones for anomaly detection". *Mobile Networks and Applications*, vol. 14, n° 1, p. 92–106.

Schreiber, Thorsten, Juraj Somorovsky, and Daniel Bumeyer. 2011. *Android binder, android interprocess communication*. Technical report.

Shabtai, Asaf, Yuval Fledel, Uri Kanonov, Yuval Elovici, and Shlomi Dolev. 2009a. "Google android: A state-of-the-art review of security mechanisms". *arXiv preprint arXiv:0912.5101*.

Shabtai, Asaf, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. 2009b. "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey". *Information Security Technical Report*, vol. 14, n° 1, p. 16–29.

Shabtai, Asaf, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. 2012. "“Andromaly”: a behavioral malware detection framework for android devices". *Journal of Intelligent Information Systems*, vol. 38, n° 1, p. 161–190.

Shon, Taeshik and Jongsub Moon. 2007. "A hybrid machine learning approach to network anomaly detection". *Information Sciences*, vol. 177, n° 18, p. 3799–3821.

Sigletos, Georgios, Georgios Paliouras, Constantine D Spyropoulos, and Michalis Hatzopoulos. 2005. "Combining information extraction systems using voting and stacked generalization". *The Journal of Machine Learning Research*, vol. 6, p. 1751–1782.

Silberschatz, Abraham, Peter B. Galvin, and Gagne Greg, 2012. *Operating System Concepts, 9th Edition*. 944 p.

Smalley, Stephen and Robert Craig. 2013. "Security Enhanced (SE) Android: Bringing Flexible MAC to Android.". In *20th Annual Network and Distributed System Security Symposium (NDSS)*.

Smola, Alex and SVN Vishwanathan. 2008. "Introduction to Machine Learning".

Song, Maoqiang, Wenkuo Xiong, and Xiangling Fu. 2010. "Research on architecture of multimedia and its design based on android". In *International Conference on Internet Technology and Applications*. p. 1–4. IEEE.

source.android.com. 2014. "Android Security Overview". http://source.android.com/devices/tech/security/index.html#android-platform-security-architecture. Accessed: 02 August 2014.

Sugar, Catherine A and Gareth M James. 2003. "Finding the Number of Clusters in a Dataset". *Journal of the American Statistical Association*, vol. 98, n° 463, p. 750-763.

Tan, Pang-Ning, Michael Steinbach, and Vipin Kumar, 2005. *Introduction to Data Mining*, chapter 8: Cluster Analysis: Basic Concepts and Algorithms, p. 487-568. Addison-Wesley.

threat description, F Secure. 2011. "Threat description trojan android.droidkungfu.c". http://www.f-secure.com/v-descs/trojan_android_droidkungfu_c.shtml. Accessed: 02 June 2014.

Tsang, Ivor W, James T Kwok, and Pak-Ming Cheung. 2005. "Core vector machines: Fast SVM training on very large data sets". *Journal of Machine Learning Research*, vol. 2005, n° 6, p. 363–392.

Tu, Zhuowen. 2007. "Learning generative models via discriminative approaches". In *IEEE Conference on Computer Vision and Pattern Recognition*. p. 1–8. IEEE.

Ulusoy, Ilkay and Christopher M Bishop. 2006. "Comparison of generative and discriminative techniques for object detection and classification". In *Toward Category-Level Object Recognition*. p. 173–195. Springer.

university of waikato, Machine Learning Group: The. 2012. "Weka: Data Mining Software in Java". http://www.cs.waikato.ac.nz/ml/weka/.

Van Lunteren, Jan et al. 2006. "High-Performance Pattern-Matching for Intrusion Detection.". In *International Conference on Computer Communications (Infocom)*. p. 1–13. Citeseer.

Vinod, P, R Jaipur, V Laxmi, and M Gaur. 2009. "Survey on malware detection methods". In *the 3rd Hackers' Workshop on Computer and Internet Security*. p. 74–79.

Wagner, David and Paolo Soto. 2002. "Mimicry attacks on host-based intrusion detection systems". In *the 9th ACM Conference on Computer and Communications Security*. p. 255–264. ACM.

Wang, Xinran, Yoon-Chan Jhi, Sencun Zhu, and Peng Liu. 2009. "Detecting software theft via system call based birthmarks". In *Annual Computer Security Applications Conference*. p. 149–158. IEEE.

Warrender, Christina, Stephanie Forrest, and Barak Pearlmutter. 1999. "Detecting intrusions using system calls: Alternative data models". In *the IEEE Symposium on Security and Privacy*. p. 133–145. IEEE.

Waseem, Fadel. 2010. "Techniques for the Abstraction of System Call Traces to Facilitate the Understanding of the Behavioural Aspects of the Linux Kernel". Master's thesis, Concordia.

Wolpert, David H. 1992. "Stacked generalization". *Neural networks*, vol. 5, n° 2, p. 241–259.

Wu, Sun, Udi Manber, et al. 1994. *A fast algorithm for multi-pattern searching*. Technical report.

Xie, Liang, Xinwen Zhang, Jean-Pierre Seifert, and Sencun Zhu. 2010. "pBMDS: a behavior-based malware detection system for cellphone devices". In *the third ACM conference on Wireless network security*. p. 37–48. ACM.

Y. Ng, Andrew and A Mihael I, Jordan. 2002. "On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes". *Advances in neural information processing systems*, vol. 14, p. 841.

Yu, Qian, Thang Ba Dinh, and Gérard Medioni. 2008. Online tracking and reacquisition using co-trained generative and discriminative trackers. *Computer Vision–ECCV 2008*, p. 678–691. Springer.

Yuan, Fangfang, Lidong Zhai, Yanan Cao, and Li Guo. 2013. "Research of Intrusion Detection System on Android". In *IEEE Ninth World Congress on Services*. p. 312–316. IEEE.

Zhang, Weizhe, Yuanjing Zhang, Hongli Zhang, Xuemai Gu, and Albert MK Cheng. 2009. "A Memory-Efficient Multi-pattern Matching Algorithm Based on the Bitmap". In *Fourth International Conference on Internet Computing for Science and Engineering (ICICSE)*. p. 1–5. IEEE.

Zhao, Min, Fangbin Ge, Tao Zhang, and Zhijian Yuan. 2011. Antimaldroid: an efficient svm-based malware detection framework for android. *Information Computing and Applications*, p. 158–166. Springer.

Zhou, Wu, Yajin Zhou, Xuxian Jiang, and Peng Ning. 2012. "Detecting repackaged smartphone applications in third-party android marketplaces". In *the second ACM conference on Data and Application Security and Privacy*. p. 317–326. ACM.

Zhou, Yajin and Xuxian Jiang. 2012. "Dissecting android malware: Characterization and evolution". In *Security and Privacy (SP), 2012 IEEE Symposium on*. p. 95–109. IEEE.