

Università degli Studi dell'Insubria

Facoltà di Scienze MM.FF.NN.
Corso di Dottorato in Informatica
XXVII CICLO



A Real-Time Framework for Malicious Behaviour Discovery on Android Mobile Devices

A Ph.D. thesis presented by

Marco Taddeo
610214

Advisor: Alberto Trombetta
January 2015

to Simona: you will be forever my "cu"

to my parents: I promise, this is my last thesis

to my friends: this time it's up to me to offer

Acknowledgements

First and foremost, I would like to sincerely thank my advisor Professor Alberto Trombetta for the constant support, collaboration and infinite patience. During the last years, as MS Thesis advisor first and as Ph.D. advisor later, he helped me with precious suggestions.

I am also really grateful to Dr. Igor Nai Fovino for his help and collaboration in many works done together.

Moreover, I wish to thank all my DISTA colleagues for the tips and the advices.

Abstract

Android is an operating system based on the Linux kernel and currently developed by Google. In few years since its first public version, Android has become the most widespread operating system among mobile devices. During this time Android has seen a number of updates to its base system. These updates added new features and possibilities but have also increased complexity and hardware requisites. Most of all, this continue evolution has caused a deep fragmentation of the Android ecosystem with up to sixteen different versions simultaneously present on the market. While the newer versions include bug fixes and critical updates to the known problems, the oldest ones remain on the market for years without further corrections exposing the system to a whole series of attacks.

Then, due to its extreme popularity and the personal information contained on smartphones - as financial accounts, private photos and other acquaintances' data - Android has captured the attention of many criminal organizations and hackers which could use the known bugs to steal information or use the device for malicious purposes. The consequence is the massive presence on the market of malwares targeting the Android architecture. Normally these malwares are popular applications in which a malicious piece of code has been inserted by performing a reverse engineering on the available application package or by inserting directly the malware if the application source code is released. The phenomenon is far more pronounced if the user downloads the application from an alternative market or pirate site to not pay the requested application price.

A great amount of research has focused on mechanisms to discovery such kind of attacks and specific Android threads analysing the applications as a whole package before installing them on the device, looking for common patterns and specific features as the use of API targeting personal information or the presence of connections with external unknown websites. These solutions are valid but unusable directly on the phone by common users. Further an application could download additional code after the installation and then any a priori control could be bypassed. Again, to discover a specific patter it must have been seen previously which is not feasible for zero day attacks. Other approaches work a runtime trying to discover the infection on the phone during the attack itself. Such methods require high computational power which penalizes the performances and battery autonomy. To be effective, these approaches require to know which the malware behavior is to be able to classify it correctly. With hundreds of different malwares available on the market it is impossible to classify them all and have specific signatures. Everything is complicated by the fact that a malware could hit most parts of the system simultaneously.

In this thesis we present a novel framework for runtime monitoring the Android

device's behavior without compromising the user experience and device computational capacity (ARAM framework). Our approach, thanks to a client-server architecture, permits to know in time many information related to the phone and the applications running on it. On the mobile device, a specific application, which works in background, collects as many information as possible both regarding the phone itself - cpu utilization, ram utilization, battery remain percentage, number of Kbyte of data sent and received, number of calls performed, SMS sent/received etc... - and the installed applications - developer information, required permissions, package signature, package hash, data sent and received for single application, etc. All these information are send to a remote server which collects and analyses all the information. If the information are considered harmful and symptomatic of an infection then the user is alerted with a message containing the cause of the abnormal event. The server offers different modules to analyse the gathered data. Each module has a specific function and interacts with the user in a different way. All of them try to minimize the user interaction with the phone to not affect what is the real use of the mobile device. All the computations and controls are performed on the server directly and then no computational power on the device is required. All the process is completely invisible to the user. The only exception is, of course, if an infection is discovered.

The first module offers a manual tool to analyse the data acquired from the devices. With the tool the user can see graphically his device behavior over time and compare it with other logs. The logs comparison permits to underline the difference among intervals of time as could be the number of SMS sent or the different battery duration with a different pool of installed applications. Further, this instrument is designed to help the user (and the system administrator) to better tune up the other modules.

The second module permits to define behavioural rules through an ad-hoc language. Each rule is validated while the data is collected and thanks to means and averages the system is able to automatically control the device's behavior in time and understand if it is the result of an infection. For example it is possible to define rules which check the number of SMS sent over time and alert the user if the current behavior is different from what has been registered previously. In the same way it is possible to define rules to understand if the phone is performing abnormal computations while it should does not (as when it is in charge or in standby). By combining these rules is possible to have a strong control over the device.

The third module analyses all the applications installed by the user and with the contribution of the data collected from the other users and the Google Play Store, it is able to understand if a fresh installed application is different from what could be considered its "safe" version. The "safe" version could be the application available on the Google Play Store or the one most diffused over the ARAM community. If an application installed by the user differs from the safe for the permissions required, the signature, the hash or the amount of data it generates over time then there is the possibility of a tempered application and the user is alerted. The user is also able to define his own behavioural rules to control its device and better meet his habit and requirements. To our knowledge we are the first to present a mechanism to define and control Android behavior through the definition of rules. During our tests we were able to discover if an application has been infected with the introduction of a

malicious code and to understand if the device behavior deviates in time in respect to the user standard profile which was built dynamically over time. Any malwares which steal information and generates data over the available internet connection is easily discovered with enough data to analyse.

The last module has a completely different purpose in respect to the first three proposed. The data acquired in time is used to reinforce the user login process on thirty-party websites. Every time the user wants to perform an action on a website (as could be to upload a photo on Facebook or to post a new comment on Twitter) the device send with the request also a “snapshot” of the device performed at the request time. This snapshot is a collection of information which are forward by the third-party website to the ARAM server. This last checks if the collected information are congruous with the user profile. Only if the information are in line with the user profile the Facebook/Twitter server performs the user request.

In the future, with the diffusion of Android wearable objects such as watches and glasses, the personal data acquired from users and environment will be greater, more accurate and therefore with a greater chance of discrimination. This will permits to our module to offer further possibility and reinforcement capacity.

In conclusion, the overall system effectiveness is the result of the contribution of each module. In fact, each module has a different purpose and it is able to discover a specific malware attack. While a single module could fail and not be able to readily discover a malicious activity, the multi-control approach ensures a higher discovery capability with less chance of error. From our tests our architecture was able to discover the most common malware targeting the Android ecosystem.

Contents

1	Introduction	11
1.0.1	Introduction	11
2	Android: The Operating System	14
2.1	Android: The Operating System	14
2.1.1	The System Structure	14
2.1.2	The Application Structure	16
2.2	The Security	18
2.2.1	The Sandbox	19
2.2.2	Secure Processes Intercommunication	20
2.2.3	Application Signing	21
2.2.4	Permissions	21
2.2.5	Manifest File	22
3	Android: The Security Problems and State of the Art	23
3.0.6	Permissions model	23
3.0.7	The SandBox and Linux permissions	26
3.1	State of the Art	28
4	Android: A Malware Attack Example	32
4.0.1	The idea	32
4.1	Malware Implementation	33
4.2	Malware Impact on System Performance	36
5	The ARAM Project: The Architecture	39
5.0.1	The Main Idea	40
6	ARAM: Snapshot Module	44
6.0.2	ARAM's Client	45
6.0.3	ARAM's server	53
6.0.4	Hardware	55
6.0.5	ARAM's Client Benchmarks	55
6.1	Malwares Implementation and Analysis	57
6.1.1	Hypnotoad	58
6.1.2	Caller Details	59
6.1.3	Battery Widget	60
6.1.4	Ministock	60
6.1.5	24h Analog Widget	60

6.2	Analysis Software Introduction	61
6.2.1	SimpleAnalyzer Tool	62
6.2.2	Oscillation Coefficient	63
6.3	Test Samples	63
6.3.1	Battery Widget Data	63
6.3.2	Caller Details	66
6.3.3	Hypnotoad	69
6.3.4	24h Analog Clock	71
7	ARAM: Policy Module	83
7.1	The Overall Architecture	83
7.2	Policies classes	84
7.3	Policies format	85
7.4	Implementation	90
7.5	Further tests and considerations	94
8	ARAM: Application Module	96
8.1	The Idea	96
8.1.1	Confrontation Algorithm	100
8.1.2	Applications Profiles	100
8.2	Approach Limitations and Heuristic Analysis	101
8.3	Tests	103
8.3.1	Google Play Store	103
8.3.2	HAA	106
8.3.3	New policies	110
9	ARAM: Identity Reinforcement Module	112
9.1	The Idea	112
9.1.1	Requisites and Constraints	113
9.2	User Profile and Confrontation Algorithm	114
9.3	Design and implementation	121
9.4	Tests	122
9.4.1	Legitimate Snapshots	122
9.4.2	tampered Snapshots	124
10	Further Data Mining Studies	125
10.1	Identity Reinforcement	125
10.1.1	Tests results	126
10.2	Malware Discrimination	127
10.2.1	Tests results	128
11	Conclusion	130

List of Tables

6.1	ARAM SQLite DB	46
6.2	Battery Widget global CPU usage graph	64
6.3	Caller Details CPU and RAM comparison	67
6.4	Caller Details Data Traffic	69
6.5	CallerDetails number of calls	69
6.6	Hypnotoad CPU values	70
6.7	24h Analog Clock miner CPU values	72
6.8	24h Analog Clock sound recorded CPU values	73
6.9	24h Analog Clock constant recorded CPU value	75
6.10	24h Analog Clock silent calls CPU values	77
6.11	24h Analog Clock number of calls	79

List of Figures

1.1	2014 Worldwide devices shipments by operating system (thousands of units and percentage)	11
1.2	2014 Worldwide smartphones sales to end users by operating system (percentage)	12
1.3	The deep fragmentation among Android devices [33]	12
2.1	Android's software stack	15
2.2	Application Lifecycle	17
4.1	Overall benchmark scores from AnTuTu	37
4.2	Detailed benchmark scores from AnTuTu	38
4.3	Time to retrieve and save the user information from the dump	38
5.1	ARAM overall architecture	40
5.2	Four module communication	41
6.1	The Snapshot Module scheme	44
6.2	ARAM's client	49
6.3	Client registration form	50
6.4	Client general interface	51
6.5	States counter	53
6.11	Battery Widget global CPU usage graph	65
6.12	Battery Widget global RAM usage graph	65
6.13	Battery Widget global battery usage graph	66
6.14	CallerDetails Global CPU usage graph	67
6.15	Caller Details Global Battery usage graph	68
6.16	Hypnotoad CPU graph	70
6.17	Hypnotoad global battery usage graph	71
6.18	24h Analog Clock miner global CPU usage graph	72
6.19	24h Analog Clock miner global battery usage graph	73
6.20	24h Analog Clock sound recorded global battery usage graph	74
6.21	24h Analog Clock sound recorded battery usage graph	74
6.22	24H Analog Clock constant sound recorded global CPU usage graph	76
6.23	24H Analog Clock constant sound recorded Battery usage graph	76
6.24	24h Analog Clock silent calls sound recorded global CPU usage graph	78
6.25	24h Analog Clock silent calls recorded battery usage graph	78
6.6	Daily average page	80
6.7	Admin server side graphic interface	81

6.8	Antutu benchmark with different acquisition intervals	82
6.9	Battery duration with different acquisition intervals	82
6.10	Overall traffic generated over 24 hours	82
7.1	The Policy Module architecture	83
7.2	The insertion form	91
7.3	Bob's last seven snapshots	92
7.4	Latest snapshot is validated as normal	92
7.5	New snapshot has screen off and CPU and RAM at very high percentage levels	93
7.6	New snapshot validates the policy	93
7.7	Error propagation on the ARAM client	93
7.8	Snapshots validation history page	94
8.1	Application Module overall architecture	97
8.2	Status icons	98
8.3	Application Module GUI with the applications list	99
8.4	HAA icons	102
8.5	Facebook information after the APK installation	103
8.6	Information related to the two app profiles	104
8.7	Facebook information after the ARAM server validation	105
8.8	AnagramSolver after the installation and before the validation	106
8.9	AnagramSolver after the validation	107
8.10	Andor's validation permits to discover some discrepancies in the permissions list	107
8.11	Situation after the CoverMe application has been installed	108
8.12	The ARAM Server wasn't able to find CoverMe on Google Play Store	108
8.13	The ARAM Server needs more information to take a decision	109
8.14	The user's app is considered as tampered with HAA enabled	109
8.15	The user's app is considered as the safe after more data is gathered	110
8.16	Traffic generated by the application in different devices	111
9.1	Identity Module Architecture	114
9.2	Pressure Values Reordered on a Client	120
9.3	Requests Valuation Response	123
9.4	Requests Valuation Response	124
10.1	A malware can alternate its behaviour in time	129

Chapter 1

Introduction

1.0.1 Introduction

According to the last available statistics from Gartner [42][44], the shipping of mobile devices has quickly increased in the last 10 years and have stabilized in the last couple of them due to market saturation. In the year 2014, 1,862,766,000 mobile devices were shipped while if we consider the worldwide device shipments by operating system (which include desktops, notebooks, tablets, etc.) 2,432,927,000 units were sent and 1,168,282,000 of them used Android as operating system as shown in Fig.1.1.

As consequence, Android is the most diffused and used operating system with a market share of 48% as shown in Fig.1.2. Since its first commercial version in the 2007, Android has seen a number of updates to its base system. These updates added new features and possibilities but have also increased complexity and hardware requisites. Most of all, this continue evolution has caused a deep fragmentation of the Android ecosystem with sixteen different versions simultaneously present on the market as is possible to see on Fig.1.3.

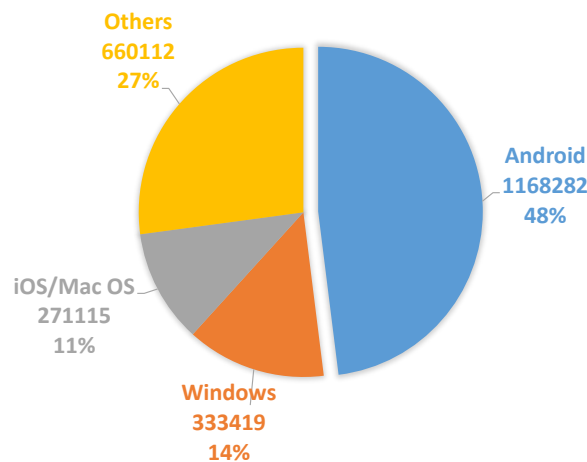


Figure 1.1: 2014 Worldwide devices shipments by operating system (thousands of units and percentage)

This fragmentation has origin due to three main causes:

- Manufacturer: rarely a manufacturer updates its phones' software during de-

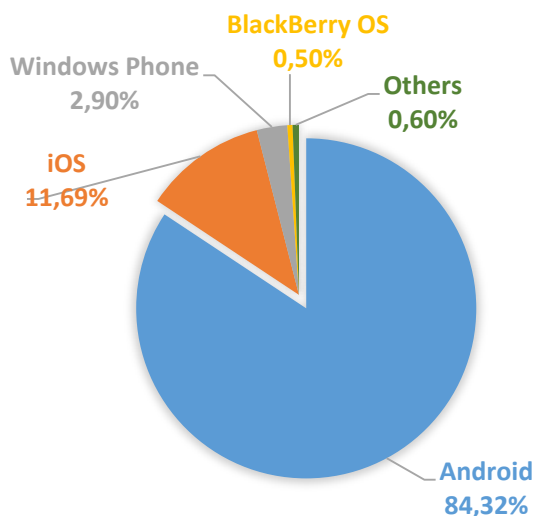


Figure 1.2: 2014 Worldwide smartphones sales to end users by operating system (percentage)

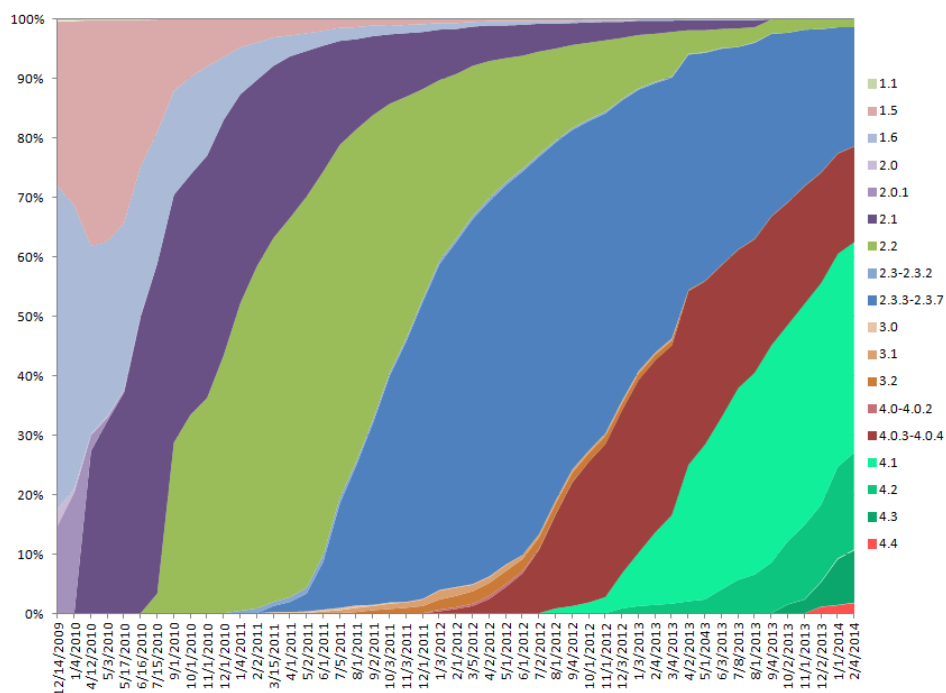


Figure 1.3: The deep fragmentation among Android devices [33]

vices lifetime. The reason is quite simple: to contain costs and force customers to buy new devices.

- Developer: Google doesn't directly manage the updates to its operating system but leaves to the manufacturers and to the mobile network operators manage how to they personalize the system and handles the updates. With the high concurrency present on the market all the efforts are used to design new models

and services leaving the old smartphone as they were when shipped from the magazine.

- User: customers can get exactly the phones they desire - with big or small display, cheap or expensive, with any number of different features combination. Market is than invaded by hundreds of different products just to satisfy as many customers as possible. Also users normally don't change their smartphones until a true reason force them to buy a new model. Then a phone can stay on the market and still be used by many people for 3-5 years without get any update.

This fragmentation will rise in the future because most of the users will not change their devices with newer just to follow the software exponential grow.

To understand how fragmentation is also a security problem we need first to understand how Android is designed and how its security mechanisms work . Also its great diffusion, with the terrific numbers we saw before, has attracted the attention of criminal organizations and malwares developers. According to a thread report from Fortune [34], 97% of the mobile malwares are targeting Android platforms.

Up against all these security issues we will present ARAM - acronym of Android Real-time Analyzer Monitor - which is a complete framework designed to monitor any Android devices and prevent the most diffused malware attacks available currently on the market.

The thesis is so structured: second chapter will introduce Android itself. Third chapter will cover Android security mechanisms, known attacks and related work. These two chapters are fundamental to understand ARAM motivations and design. Chapter four is dedicated to a specific Android memory issue with the development of a malware targeting such weakness. Chapter five introduces ARAM and shows it goals and overall architecture. Chapters six, seven, eight and nine present ARAM's components in details (the modules) specifying which is the contribute of each of them. Chapter ten presents a further data mining on the data collected using the Weka library. Chapter eleven concludes the work.

Chapter 2

Android: The Operating System

2.1 Android: The Operating System

Android is an open source operating system designed for a wide range of mobile devices (smartphones and tables are just the tip of the iceberg). It is based on the Linux Kernel and it is currently developed by Google which releases Android's source code under open source licenses. In this overview we will cover the technical aspects of the operating system. These information are fundamental to understand the choices that we have undertaken while developing the ARAM framework.

2.1.1 The System Structure

Android's structure can be abstracted as three main blocks:

Device Hardware: Android runs on wide range of devices, from cellphones to tablet, set-top-boxes and shortly on cars and televisions. It is almost indifferent which processor is utilized, because Android works fine with all of them (MIPS, ARM, x86, x64, etc.);

Operating System: Android OS is entirely built on Linux kernel. All sensors and device resources such as telephony functions, GPS data, Bluetooth and other network connections, camera functions... can be accessed only through the operating system;

Android Runtime Application: This section includes all the applications run within Android. Often, these applications are written in Java programming language and runs in Dalvik Virtual Machine. But some of these, such as Android core services, can be native applications or include native libraries. However both Dalvik and native applications run in sandboxes to increase security level. That means that applications run inside a dedicated part of the device and each of them cannot leave its area. Then, even if something goes wrong in an application, the system continues to work.

This list can be sharpen to get more details about how the platform works. We can divide it in 5 principal groups:

- Linux Kernel;

- Android Runtime;
- Android Extra Libraries;
- Application Framework;
- Applications.

The structure schema is summarized in Fig.2.1

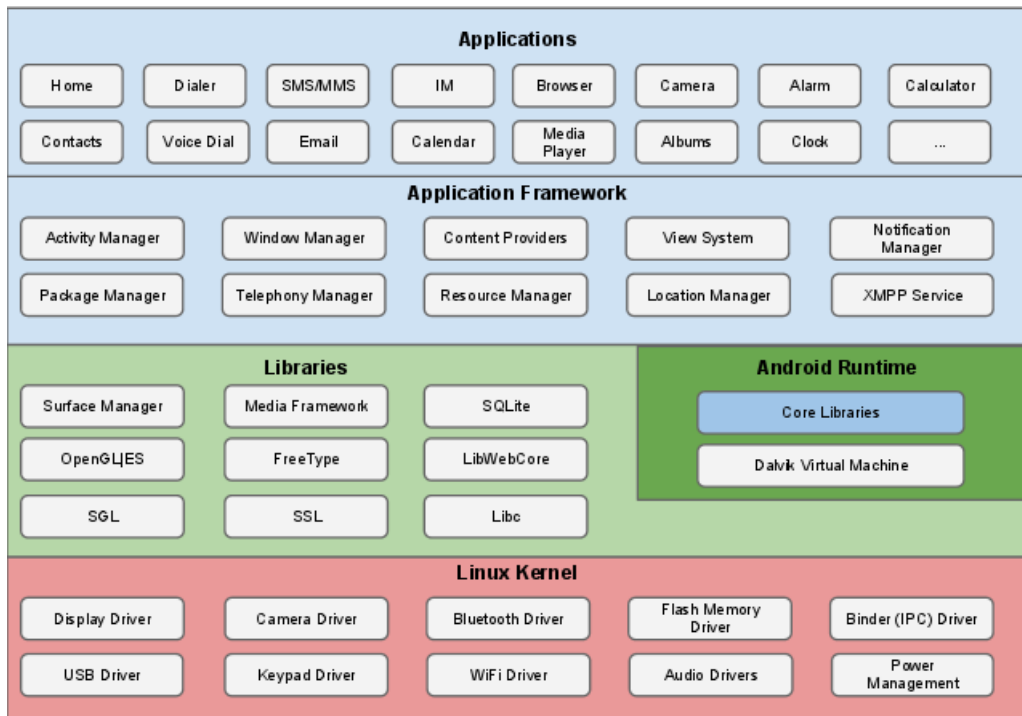


Figure 2.1: Android's software stack [32]

As we can see, there are 4 principal levels. On the bottom we find the Linux kernel's modules, which guarantees the driver compatibility with all the hardware of the device, such as camera, flash memory, bluetooth, Wi-Fi, display, etc. Also, it is responsible of the power management, which is extremely important aspect on every mobile devices.

In the higher level we found both Android Runtime and Android Extra Libraries. The first one contains core libraries and the Dalvik virtual machine, which will be soon substituted with the ART virtual machine. Instead the Android Extra Libraries group contains codes of frameworks and libraries used by the upper levels. In this list we want to stress the SSL library, used for security communications, but also SQLite which allows applications to easily storage data.

The Application Framework level is responsible of the control of the actions done by the applications. For example is not possible for an app to locate the device without using the Location Manager service provided by this level.

At the top level of this stack, we can find the Application layer. All the applications developers do reside in it. And all those we can find in the official Google

Play market, once installed, will be inserted in this layer. In this way, applications cannot directly access to sensible information.

2.1.2 The Application Structure

Android applications are mostly written in Java programming language, but they also may contain C and C++ codes.

Android applications are a collection of files compiled by the Android SDK in a special archive which extension is *.APK*. We will refer to these archives as just "APK" during the thesis. The APK contains all the classes and the application resources and it is the format in which the applications are distributed among the market.

Any Android application has a precise internal structure:

src/ this folder contains the Activity files and all the Java classes;

bin/ here we can find the final *.APK* file and all the other compiled codes;

jni/ contains all the source files in C and C++;

gen/ this contains the Android auto generated files, such as R.java;

assets/ this is an empty folder where developers may store raw asset files which will be compiled into the *.APK* file as-is;

res/ contains the application resources in a sub-tree of folders we do not show in this thesis. In this folder we can find, for example, the layout of the application;

libs/ here are stored all the external libraries the application may needs;

AndroidManifest.xml This file contains some important information about the application, such as the permission the application needs.

In the root directory we can find also other files, but they are not fundamental for our scope.

Each application is composed by different block which may communicate between them using intents. The possible blocks are four:

Activities: An activity is represented by a single screen application with a user interface. For example, we can have an activity for receive new mail, another for read new mail and another one for write a mail message. All these work together for create a complete application for manage email;

Service: A component which runs without user intervention in the background and performs long time running operations. Services do not provide any user interface. For example, playing music while user runs other applications requires the use of a service. Services can be started by other components, such as activity. They can be run free or can be bound to it for achieve interaction with the service;

Content Providers: This component manages shared applications' data. Through a content provider, other applications may read and modify those data;

Broadcast Receiver: This component is responsible to respond of system announcements. The operating system provides a broadcast message through all the device for example when the battery is low. A broadcast receiver deals with these messages and executes some developer's code as answerer to that state. Broadcast receiver do not provide a properly user interface, however they can show a status bar notification or a Toast (pop up) message.

Android is designed in a unique way. An Android application can start another application's component. For example, it can start the component of another application which capture a photo an import the obtained image in its own component. Instead of developing another activity to do this, we can simply call other application public functions. The Android apps do not contain any *main()* method and any component of these is accessible from other applications. However, an application could not have direct access to other application parts because the system runs each application in a dedicated sandbox.

When an application runs on Android, it is confined in a sandbox defined by the operating system. Applications follow a complex life cycle we can see in Fig.2.2.

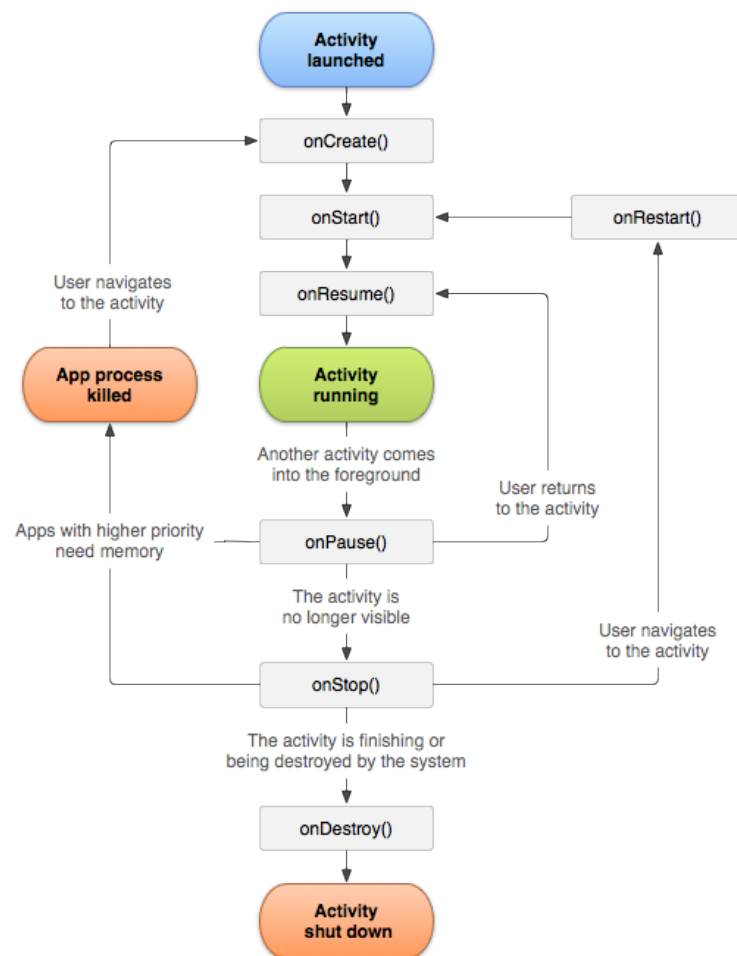


Figure 2.2: Application Lifecycle [30]

What interest us is understand how the operating system manages the applications lifecycle. This will help us to understand how to create our monitoring application and the malicious codes to test it.

2.2 The Security

Android is a platform with an high level of default security, but unfortunately these security systems are not enough to protect our information, as we show later. However for now we want to present how Android implements its own security services.

According to the description of the security of the Android operating system [32], Android is the most secure operating system for mobile devices. It is built to protect user data, protect system resources and provide application isolation. To obtain these objectives, Android use the following security features:

- Security at the Operating System and Kernel level;
- Mandatory sandbox for each application;
- Secure interprocess communications;
- Application Signing;
- User granted and application defined permissions.

The basis of the Android system is the Linux kernel. This is used with success in millions of security sensible environments. It has been targeted of several attacks during its history, and for every security breach there always has been a security update. This brings Linux kernel to become a stable and secure system, trusted by numerous corporations and professionals. This result can be achieved through the structure of Linux itself. It presents many internal security systems which grants security of data and access controls. It uses:

- User based permission model;
- Process isolation;
- Mechanisms for secure interprocess communications (IPS).

But what makes of Linux kernel the perfect candidate for mobile devices is its multiuser structure. In fact Linux is built to separate file and resources for each user using it. This feature is perfect for isolate applications from each other.

Linux's security policies provide:

File security: A user cannot read file of another user. Then user *A* cannot read, write nor execute files of user *B*. In Android that means that there is not the possibility for an application to access data of another one;

Resources Management: Linux provide a solid structure for the memory, processors and hardware components. No user can exhaust the memory of another one. Then user *A* cannot access to the memory assigned to user *B*. Also it impede to a user to take some memory forever. Similarly, Linux manages processors so that no user can access data nor resources itself if it already in use

by another one. It can guarantee also that user A do not exhaust B 's CPU. In Android this translates with applications separation. Also, this can guarantees that if an application crashes, the system can continue to work. And the resources that application has locked eventually will be released to the other one who needs it. So, Android intervenes if an application that locks GPS do not unlock it correctly, and assign that resources to another who has requested for it.

The Android OS presents two powerful security systems. First of all, data and systems are maintained in two separate partitions. In the system one we can find the Android's kernel and the operating system in addition to applications, application runtime, application framework and system libraries. Also this partition is read only by default, and nor users neither applications can write in it without root permissions. As we will see later, only a small portion of code in Android platform runs with root permissions. Android adds a second useful security system at OS level. It permits to users to boot their devices in Safe Mode. This means that all third party software will not be loaded, so the system runs with only Android's core applications. This transforms the phone in a completely safe environment form which users are able to remove any software problems.

2.2.1 The Sandbox

To achieve the applications isolation, Android use a method known as sandbox. When an application starts, Android system assign to it an unique user code (UID). Using it, Android treats each application as Linux's user. In this way, Android can easily exploit Linux's resources management mechanism. This approach has never done before in an operating system. Linux itself do not use it in this way. In fact, In Linux we have a user which runs multiple application within its user's space.

In Android things are little different. Each application is considered as a Linux's user and Android maintains them separated from each other and from the system. However, communication between applications and operating system are possible but these occur through user permissions. For example, if application A tries to do something malicious like read application's B data, the operating system blocks this action because A does not have the necessary user privileges. In addition to this, we have to say that system's applications, like make a phone call, are treated the same way. This means that an application without the appropriate permissions cannot do a system call, like dial phone. System's call application are mostly showed in the Application Framework level of image 2.1. Since Android's sandbox structure is implemented in the kernel level all the codes in levels above it runs in a sandbox. That include all the native code and the operating system applications. With a so solid architecture, it is also not necessary to specify a developer have to write his/hers own application to make it secure. Android is the only operating system now which does not require a further specifications to enforce security of an application. The sandboxes guarantee a more solid system. In fact, a memory corruption of an application involves the compromising of the entire system, in a normal mobile OS. Instead, in Android a situation like this leads to the corruption of the sandbox only. And the arbitrary execution of code that could results is limited to the permissions operating system give to that application. The only applications that do not be part

of this description are the ones with root permissions. These applications may run outside a sandbox and have permissions to read, write and execute each files in the Android's device. This implies that rooted applications have complete access to each other application, and to their data. However, by default, only a small set of applications that belong to the core code and the kernel have this permissions. Android prevents that applications gain this kind of permission because an application with it can compromise the integrity of the entire system.

There are many methods to obtain the root permission. This ability causes exposure to malicious codes increasing the likelihood of malwares' infection. However, this feature in the Android system may be useful for developer users. In fact, there is the possibility to install system components not present by default in an Android device which may be helpful for optimizing the OS on specific hardware. Also, installing application with root permission may be used to add some applications and features not provided by Android. The default method to obtain root permission comports the installation of a new operating system which provides it. Android protects existing user data from unauthorized installation of a software like this, inserting the cancellation of all user data as part of this process. However, there are other methods which use an exploit in the kernel or a security hole. With these is it possible to obtain root permission bypassing the cancellation of user's data. We want to stress that encrypting data with a password key does not protect application's data from a rooted app. Neither if the key is stored on the device nor if it stored somewhere else, such as an external server. In fact, in the first case a rooted application may reads directly that key. In the second case, we can say that the application which encrypts its data reads the key, eventually. When it happen, the application with root permission may read the key and gain complete access to all encrypted data. Here, encryption may just temporary protects data in the best case.

2.2.2 Secure Processes Intercommunication

Android Sandbox keep applications separated. However, sometimes it is necessary for an application to send or receive data for complete its job. It is not always possible to do all the computation requested without cooperate with other processes. For this reason, Android provides a security system for the maintenance of secure communication within processes (IPC). We already said how important it is to keep applications separated but it is also necessary to make communications between applications. Android provides all the Linux traditional mechanisms which include sockets and signals. It also implements some new security communications methods:

Services: processes which can perform background and long-term executions. Services are extremely powerful because they can also bind to other processes and perform interprocess communications through binder. For example, a service may manage network transactions, read and write files, interact with other applications... all in the background;

Intents: simple passage of an application "intention" to do something or the broadcasting of interesting events. For example, an application may want to execute the browser and open it to a specific page. The application need to specify its intention to show that web page creating an intent and sending it to the operating system. It identify which portion of code the intent needs, and execute

it. Otherwise, an intent can be used to send a notifications system-wide. For example, when the internet connection got lost a broadcast intent is sent to all applications which needs it;

Binder: it is a basic call mechanism which performs in process and interprocess communications. It is the base class for remote objects and it is implemented from custom Linux driver. For example, services provide interfaces accessible through binder.

ContentProviders: this is a method to expose data of an application and also to retrieve those from other processes. For example, to store the list of contacts of an user, Android use a ContentProvider. Application which want to access these data reads that ContentProvider to obtains those. Application can also define its own ContentProviders and expose data for other applications.

Android team encourage the use of these methods instead of the UNIX-type ones. Even if Unix systems are solid, best practices recommends the use of the Android version because in this way developers cannot add involuntary security flaws.

2.2.3 Application Signing

Android provides API and methods to sign the applications. The signature is required for any code that may be installed on the devices. In fact, both Google Play and package installer on Android reject applications which do not have any kind of signature. Sing an application guarantee to Android which developer, or developer team, has done the application. Also, it guarantee to the developer that him/hers application is not altered by third-party. This procedure is also the first step in the applications isolation paradigm. The signature certificate defines the possible iterations between applications. Applications can communicate each other only if they share the same certificate. Otherwise applications cannot communicate with each others without the use of an IPC. When an application is loaded to an Android device for the installation, the Package Manager verify if it has been signed correctly, or if there was some alteration of the application. In these cases the installation fails and the application will not be installed. It is important to say that applications can be signed by third-party operators. Developers can self-sign their applications without any external verification entity. That means that currently Android does not have any Certification Authority where developers have to authenticate themself.

2.2.4 Permissions

Applications in Android have permissions to access to a strictly limited amount of resources of the device. It use the principle of least privilege. This means that each applications has access only to the minimum amount of components and to all the resources which users grant to it. Android structure as we saw in the Fig.2.1, have numerous APIs which are available only through the operating system. Some other resources, instead, are maintained secure through the intentional lack of a specific API. For example there are no API in Android which permits to manipulate SIM's data. For those resources which may be available for applications, Android request the user's consent. This mechanism is known as Permission method. According to

Android permissions web page, actually there are 146 permissions. Each application have to specify in an internal file, the Manifest file, which permissions the application needs to work. If an application requires access to a resource which is not present in the Manifest file, the resource will not be given and the application will crash. When an application is installed, and the user grants all the permissions requested, those remain granted until the application is removed. No further requests will be prompted.

2.2.5 Manifest File

We mentioned earlier the Manifest Android file. This file is mandatory for each application and has a specific name: *AndroidManifest.xml*. Further, it must be in the root directory of the application. The manifest contains essential information about the application such as the permissions it requires. But this file contains also other information:

Package name: It set the package name for the application which is used as unique identifier for the application itself;

API level: It determine which minimum API level must have the device's operating system in order to run this application;

Components description: It describe the internal components of the application and names the classes which implements these, such as activity, service, broadcast receivers and content providers;

Processes components: It shows which process will host which application components;

Permissions: It declare which permissions the application need to work; it is important to say that there is no limit to the number of permissions an application may require;

Other Permissions: It determines which permissions other applications must have in order to interact with this one;

Library: It lists the external libraries this application is linked to.

This file is extremely important and we will see how while analysing some Android security aspects in the next chapter.

Chapter 3

Android: The Security Problems and State of the Art

The permissions model, the Sandbox paradigm and the Android APK signature are strong mechanisms on the paper but have some limitations which could expose the system (and the owner) to malwares and dangerous applications behaviour.

3.0.6 Permissions model

First of all, the permissions model (System Permissions) is granted just before an application is installed. The system reads the application manifest and prompts to the user the complete permissions list and asks him to agree with them to continue within the installation. If the user agrees and grants all these permissions then the system completes the installation. The user cannot choose to grant or deny a specific permission but must grant or deny them all. If user chooses to not grant the entire set then the installation is aborted.

Currently, with Android KitKat 4.4, there are approximately 130 different permissions. While this list could be easy to understand for an IT specialist or a skilled user, it is really difficult for a normal user. This typology of users normally agree without understand clearly what they are doing. They want just to complete the installation procedure and use the application. In addition to, most applications are so angry of permissions that they require as many permissions as possible just to cover any future update which may require new permissions to allow new features.

The user will never know when an application will perform an action related to a permission. For example, if the application asks for the ability to send SMS and the user grants that, he will never know if and when the application sends a SMS. Indeed, no notifications are displayed and the user just know that sometime in the future an SMS could be send.

Another problem is related to what the permission is used for. If an application requires the permission to send data over internet, no mechanisms are implemented to know exactly what the application will send during its lifetime. These data could be personal information, contacts number or any other private information which the user may not want to share to others.

For example, Twitter, WhatsApp and many other applications on the Google Play Store verify the user identity by sending a SMS (and then the user phone

number) to their remote server. To be able to do that they requires, during the installation, the ability to send and receive SMS. No one can guarantees that no further information are contained in the SMS or that no further SMS will be sent in future.

Some permissions are too course. For example, "Read Phone State and Identity" permission is needed to detect when a call is being received (for example to know when to stop a song reproduction if the app is a music player). But the same permission allows the app to harvest the IMEI too.

Thus, the user can only trust these applications and hope that their behaviour will never be malicious because with the Android mechanisms there are no further guarantee on how these granted abilities would be used for.

A common mistake is to consider secure any application available on the Google Play Store and to install it without further attention. This assumption is wrong. In fact, any developer can upload an application on the market just paying a one-time fee of \$25 dollars. Google performs some basic static controls over the application's code and could not know what an application will do at runtime and for which purposes it asks for permissions. A criminal organization could then publish on the market any kind of application just following some basic rules. Accordingly, the permissions model doesn't prevent the user to download and install malicious applications or malwares and even to benign applications to collect private information abusing of their permissions.

Google Play Store in not the only way to distribute software to the end users. An APK file could be download directly from developer website or any public server, forum or public chat. All they need to do is host the APK file and share the download link.

Some controls are performed by Google on these applications and the protection level has improved in these last years. Unfortunately these mechanism are still too weak. As we already said, users can download applications from unknown sources, and they do this really often as we can see considering the "Flappy Bird" case [3]. Flappy Bird is a popular game, which was removed from the Google Play Store because too addictive. Then, users searched for the APK on other markets. Knowing this, attackers injected malicious code in Flappy Birds and uploaded their version on unofficial stores. This way, users expose themselves to unnecessary risks.

Furthermore, as often in the computers world, many users try to download paid apps on pirate websites to not pay license fees. While famous apps downloaded from the Google Play Store are normally safe (and uploaded directly by the right developer), there are no guarantee that the pirate free of charge version contains exactly the same code as the original one. It's very easy to inject malicious code into an app, which means any app out on the wild can potentially include malicious code.

Many online guides [48, 49, 51, 52] show how it is easy to inject a new piece of code inside an application and repack it without leaving any trace. What is needed are just some free tools available online. Overall the procedure is always the same:

1. Get the original Android application APK over the internet (from the developer website or by using one of the available tools which permit to download the whole APK package from Google Play Store);

2. Decompile it using "APKTool" [45]. It is a tool to reverse engineering any Android APK and obtain "smali" bytecode [53]. It can decode resources to nearly their original form and permits also to rebuild the APK after the modifications. Originally it was not intended for piracy and illegal uses but in a while it has become the most used tool by crackers. Using APKTool we can decompile the APK and gain complete access to its resources (any files like photos, music, etc.), source code and other metadata.
3. Inject (copy) the malicious bytecode into the decompiled original application. This is another easy step: internet is full of examples. One of the most abuse malicious code tries to send SMS to premium numbers. The code is so simple that also the APK dimension is barely increased:

```
//Send SMS with text "Hi World" to phone number 555012345
SmsManager sm = SmsManager.getDefault();
sm.sendTextMessage(555012345, null, "Hi World!", null, null);
```

First the code must be compiled to Dalvik bytecode and then decompiled using APKTool in smali bytecode. Then the result can be copied inside the original decompiled APK.

4. Add any permission required by the new malicious code inside the Android-Manifest.xml file. In the SMS example, the permission required is the ability to send SMS.

```
//Enable an application to send SMS
<uses-permission android:name="android.permission.SEND_SMS"/>
```

5. Recompile the original app using again APKTool;
6. Sign the recompiled APK using "signAPK" [47]. Sign the app is mandatory since it is not possible to install an app on a device or an emulator without previously signing it.
7. Deliver the APK to the victims;

The same trick could be used to perform calls to premium number, to send email with user information to a remote server and many other attacks that rely only on permissions abuse. At the moment there are no known ways to protect against such kind of attacks.

Google developer platform, Eclipse, and the next Android Studio provide developers with the ability to obfuscate their code using ProGuard [1]. This will protect code from hackers who want to reverse engineer it starting from the bytecode. However, it is impossible to obfuscate the system calls so there will be always a possible entry point to inject code appropriately. Obfuscation has also a negative side because it prevents the possibility to easily analyze the code.

The permissions model is then a good start point but is too general and too restrictive (nothing or all).

One alternative would be to ask for permissions at runtime but the Android team thinks that this is inconvenient and insecure, so it's not likely to happen.

3.0.7 The SandBox and Linux permissions

The sandbox constraints any app, or better any app process, to rest in its specific "box" without be able to reach any other system part or resource without explicit system consent. Any file the application writes on its sandbox is private and not accessible by other processes. This security layer works until the kernel is not compromised. The only way to overcome the system is to have root access.

In the Linux (Unix) environment, the root access or root user is the highest permissions level which has the rights to perform any operation. By default, only the kernel and a small number of core apps run as superuser (normally those developed by Google itself).

If the user installs an application on a "unrooted" phone and the app requests for root permissions then the operating system prevents it without exceptions.

Everything changes if the system is "rooted". By "rooting" an Android device any restriction imposed by the manufacturer or carrier can be bypassed. This means that an application can get root permission and modify the system itself. The result is quite obvious: any security mechanism described previously is no longer effective.

Despite these security problems and all the risks, rooting is a common procedure in the Android community. Users perform the rooting procedure on their devices for many reasons, here are the most diffused:

1. Update the system: most devices are not up to date. Once a device is rooted, it is possible to manually install any update available on internet. Many manufacturers do not update their phones to the latest Android version while Google release many new and improved versions in time. These new versions contains security enhancements and bugs fix. On many websites is possible to download and manually install the new versions.
2. Install specific applications: certain apps are blocked by the carrier or the root access is required to work correctly. For example, Titan Backup [54] requires root access to perform a full device backup (in order to read any folder) while SetCPU [55] requires root access to overclock the CPU. Further, many apps are released with advertisements inside them (most of the time there are two versions of the same app - a free version with advertisements and a paid version without advertisements). This behaviour is also encouraged by Google which offer an advertisement API directly inside the Android SDK. A rooted phone permits to install applications useful to block these advertisements.
3. Remove unwanted applications: most carriers and manufacturers install own applications and graphics interfaces. These are often unwanted by the owners and by rooting the phone is possible to remove them from the system.

But applications can get root permissions also on an unrooted phone. A recently disclosed vulnerability in version 3.14.5 of the Linux kernel [57] is present in some versions of Android and this could give to attackers the ability to acquire root access on those devices. Such access could potentially allow that same attacker to run further malicious code, retrieve files, bypass third-party or enterprise security applications including containers like Samsung's secure Knox sub-operating system and establish back doors for future access on victim devices.

Find a Linux exploit is quite simple because internet is full of websites listening the kernel bugs and those information are public. Once a bug becomes known it is only a matter of time before an attacker develops an Android-specific malware.

Android has also its own specific bugs which are not inherited by the use of the Linux Kernel. Google has a public webpage where these bugs are listened and where everyone can post new bugs and track any problem related to the OS [58].

Among these Android bugs, two of the most famous are #8219321 and #9695860:

- Bug 8219321 [59][60]. It allows attackers to infect any system application (like the one developed and distributed by Google) without breaking the checks on cryptographic signature performed at APK installation. The bug itself is quite simple and it is present since Android becomes a public operating system. Android permits to have duplicate file names inside the same APK archive but, if they are present, the signature check can be tricked. In fact only the last of these homonyms is checked by the signature verifier. So while the last file could be untouched and with the correct signature, the first one could be tampered and been injected with malicious payload and not checked at all. Google released a patch to face the problem but, as we already said, most of the devices will not get the update and will always be vulnerable.
- Bug 9695860 [61][62]. Similar to the previous, it permits to edit the APK file without breaking the signature. It works by appending new code at the end of file `classes.dex`. Then, by adjusting some extra fields these new lines become invisible to the signature check.

Another interesting problem arises by combining the actions of two or more applications. Let us consider two apparently harmless applications: a ftp client and an instant messaging application like could be WhatsApp. Both the applications have "access internet", "save to internal memory" and "read internal memory" permissions. WhatsApp stores all the communications and relative media files in a local directory while the Ftp client read files from the internal memory and upload them to a remote server. Apparently, none of these can be considered as a malicious application. However the FTP client may contain a service that scans the internal memory for any WhatsApp files and send them to a server without the user consent. Such malicious behavior is not predictable by reading the permissions list. It is absolutely normal that an application which sends and receives files through internet requires network and internal memory permissions.

Then new solutions must be sought. Available security applications for mobile devices are extremely different from personal computer anti-virus. In fact on mobile devices is actually impossible to perform a full filesystem scan to try to identify malicious code. First of all because anti-virus applications in mobile devices do not have administrative privileges so they cannot do a complete analysis over the entire system. In some cases, iOS for example, the anti-virus applications, as all other app, are inserted in a sandbox by the operating system and keep separated from each other installed applications. This means that if the user wants to scan an attachment in an email he have to send the attachment to the anti-virus because operating system prevents the anti-virus to access his mail application. Other way the application needs the root permission to read other applications file. This solution is not recommended because an application with root permissions have no limitation,

and if it has a code bug an attacker might use that to take complete control of the device. For Android, the situation is similar. Applications are maintained strictly separated. In most of the cases, there is no possible interaction that allows to an anti-virus to keep an eye on malicious code, as they do for personal computers. Some of Android's anti-virus scan the phone in search of known applications that permit the root of the device. In second place, Android's anti-virus scan for known official infected applications on the phone. Then the efficiency of anti-virus software for Android depends only on the updates of the list of known compromised applications. Obviously all the zero day menaces can't be revealed this way. Also, if the list is not maintained update, the efficiency of the anti-virus decrease significantly. On top of all there are performance and battery problems: if the antivirus constantly scans memory or running processes then performances and battery are widely affected.

ARAM has been designed to face all the problems listed above. Our architecture is able to monitor the device behaviour without compromising performance and battery life. It does not base its analysis only on a priori control over the APK but, thanks to the combination of different modules, allows to remotely control the behavior of the device overcoming the limitations of other common approaches.

3.1 State of the Art

Many works cover Android security problems proposing new solutions. In this last part we will analyse those which in our opinion are the most interesting and related to our field of research. The proposed solutions are very different from each other, each with its own strengths and flaws. They can be divided into three main categories:

1. client analysis: any analysis is performed directly on the device itself. This could be achieved by a priori analysis over the APK, a runtime analysis of the behaviour or both of them;
2. server analysis: any analysis is performed directly on a server and its normally a priori analysis over the manifest and the application code. This is quite similar to the client analysis but changes drastically the computational power available.
3. client-server analysis: a set of information recorded over the phone is sent and processed on a remote server.

Sanz [16], Sato [14], Grace [6], Felt, [7], Enck [8] and Feldman [13] try to identify if an application is malicious or not by a static analysis of the manifest file contained in the application APK. Such a control could be performed directly on the phone itself or on a remote server. We consider this kind of approach inefficient. For example, we are not able to understand if an application which requires *Network Communication* and *Read Contacts* permissions is a malware or not without further investigation. An application with these permissions may be an instant messaging client like WhatsApp or a malware which steals user contacts and send them through internet. Even if we are in the presence of an instant messaging application we cannot be sure of its true nature and the malware may be hidden inside its code. With more than one hundred permissions and the "habit" to ask as many permissions as possible during the installation, it is not possible to establish a direct correlation

between the requested permissions and the application nature. Without an analysis in time there is the risk to block an harmless application while the malware could operated undisturbed. The method results too restrictive or too permissive and then it may block a rightful application while the malware could easily pass by unnoticed.

Geneiatakis [19] proposes a method to discover improper permissions utilization by monitoring when and how a granted permission will be used. Basically the method recovers the application source code by performing reverse engineering on the available APK. Then they insert special labels before any methods or API calls. Then they install the repacked application and monitor through a special kernel layer when the labels are invoked. The approach requires to break down the application and repack it with the new label in order to be able to analyses the runtime behaviour. This could be done for research purposes but not as a decisive method for the mass. Any APK must pass through a dedicate server for the reverse engineering. If the user installs the app from the Google Play Store or a third-party site than no analysis could be performed. A second related problem is the difficulty to obtain a congruous code from the reverse engineering procedure. If the code is not clean and easy understandable than is very difficult to assign label to methods and API calls. Third the system needs continuous updates to track new API and methods introduces in time. And last it requires a modification to the kernel to catch the labels. Further the special kernel layer requires to modify the system and root the phone to install it.

Shabtai [20] proposes a framework to detect malwares directly on Android mobile devices. The framework relies on an application installed on the device that samples various system metrics as the CPU consumption, the number of sent packets, the running processes and many others to train a supervision anomaly detection algorithm. All the computation are directly performed on the device. Such an approach has some limitations. First off all, to train a classifier there should be a malware sample with relative behaviour. It is impossible to train the classifier for any available malware on the market. There will be always a behaviour never seen before and for which the classifier has not been trained. However, assuming that is possible to build samples for any existing class of malwares, the discovery process would requires to tests the application with every sample. In a device with energy and computational constraints this is not feasible. Further a malware doesn't operate the whole time as a malicious entity and this complicate the training process. With a malware acting with a normal demeanour in most of the time, the training process is very difficult and almost impossible without knowing the malware nature and frequency of attack a priori. To prove their framework the authors developed four malwares and tested the classifier only with these limited malicious code. It is not known if the approach is valid for a real malware or if it is able to discover a malware which targets many system features at the same time. The lack of a centralized server that collects the data from different users also does not allow to perform cross analysis.

In Peiravian [17], Sami [18] and Wu [15] works, the malwares discovery is performed by extracting information from the application manifest and from the application dex code. These information are then use to train a classifier which tries to establish if a new unseen APK belong to the "normal" or "infected" profile. Again this approach is able to perform only a priori analysis on a dedicated server. If the user installs the APK directly on his device there no way to analyse it. Further the

application can download new code in-time by downloading it from internet and then a "normal" application could become harmful after an update.

Enck [10] proposes TaintDroid, an information-flow tracking system that permit to understand how the user's personal information are handled by the system's applications. While the methods is really efficient and lightway, it requires to insert a new layer in the Android architecture to track the information flow through the Kernel and the Userspace. This is a very strong limitation that prevents TaintDroid to be used directly out of the box. Further the TaintDroid injection requires to flash a custom-built firmware to the device, a very dangerous procedure. Another shortcoming of the method is that an application which handles user personal information does not necessarily be a malware, many normal applications with this classification approach may be considered as malwares.

Zhou [5] and Yan [4] works propose methods to monitor the application behaviour a runtime directly on the device. These approach are very effective in identifying malicious activities but performing all the computations on the device itself suffer from a significant overhead and deep battery drain.

Burguera [12] proposes CROWDROID, a server-centred architecture where a client installed on the device monitors Linux Kernel system calls and send them to a centralized server. Here the information are used to understand if the application has a malicious behaviour. The decision takes in account information sent from as many user as possible to improve the discovery rate. First of all the authors don't specify how they monitor the system calls. A normal application could not track system calls even if it has root permissions. If the application requires special library or modification to the kernel than it cannot be installed and used by all the Android community. Then again, a system call invocation is not strictly legated to a malicious behaviour. It is not possible to know if a system call has been done for good or evil operations.

Zhao [11] proposes RobotDroid, another malwares detection system based on malware signatures and machine learning approach. As in many other works, to generate valid signatures the malware must be known and act always in the same way. The approach could detect only malwares which signatures are similar to the applications already marked as malwares. If a new malware differs from the recorded behaviours then it discovery is more difficult. Also a malware could act most of the time as a "normal" application and then define a signature is again more difficult.

Arp [9] proposes DREBIN, a method to perform static analysis on the applications installed on the mobile device. The analysis collects permissions, API calls and network address directly from the application manifest and dex code. Then the dataset is used both for static and machine learning analysis. While this approach is effective, it performs only a priori analysis and base its decision only on that. Thus the method is able to discover only a subset of the malware classes for which a signature is available. Further the a priori analysis doesn't permit to know how and when a potential malicious part of code will be used. The task is far difficult if the application contains native code or can update itself at runtime from internet. Complex applications such as Facebook, Twitter and other contain thousands of API calls. In our opinion it is impossible to discriminate between these and be able to understand which are "good" API calls and which are "bad" API calls.

As we will show in the next chapters, the ARAM architecture differs from the above works for many aspects. Our approach doesn't require any root access on the device or modification to the original device firmware. Any computation is performed on a dedicated server and then the device performances are not affected. The different modules permit to perform both static and dynamic analysis. With static analysis we can discover an infected application by comparing its information with apps from the Google Play Store and from the community. This can prevent injection attacks and to discover any tampered application. With dynamic analysis we can discover a runtime any anomalous behaviour. Further, the centralized server permits to cross different users information and then alert them even before one of their applications start to act as a malware.

Chapter 4

Android: A Malware Attack Example

We spoke of bugs, exploits, sandbox, permissions and malwares trying to understand which are Android's defences and how these try to face malicious codes and hackers' attacks. But Android security problems are not just related to the operating system architecture or collection of different open source parts. Also applications developers could expose user private information, such user name and password, to possible malware attack and consequential data leak. In this chapter we will see how it is possible to retrieve very sensitive user information by installing on the device an apparently harmless application which actually hides a malware inside. First we will describe which Android part and security mechanism could be attacked to steal information and then we will build a proof of concept and evaluate its impact on the device performance.

4.0.1 The idea

Android sandbox, as we saw before, has the main functions to protect the system from running applications and to control applications interactions with the available resources and also against other running or installed applications. Each application has its own space and resources and no other application can access and read them. Every information the application ask to the user and collect for later in memory is normally available only for the application has obtained it. This memory, the Dalvik Heap, is allocated by the system every time an application is executed and its management by system routines which creates new objects and release them at application needs. As a result it contains all the variables that allocated and used during the application runtime. Its size is not fixed but follow the application requests in time.

This specific design, while is perfect on the paper, may exposes user information, saved for any reason into the heap, to be read and stolen by a specific application if it is able to break sandbox protections and scan other applications memory. Let's see how it is possible. In 2002, Michal Zalewsky develop a specific Linux tool called "memfetch" [27]. This open source code is designed, at least according to its author, to dump the memory of a program without disrupting its operation, either immediately or on the nearest fault condition (such as SIGSEGV). It can be used to examine

suspicious or misbehaving processes on your system, verify that processes are what they claim to be, and examine faulty applications using your favourite data viewer so that you are not tied to the inferior data inspection capabilities in your debugger. Technically speaking, the code access the memory of a running process using the information available in `/proc/pid/maps` (where PID is the process identification number we want to investigate). Then using these information with "ptrace" [28] it is able to copy the process memory. The best part is that the reading procedure doesn't destroy the process (memfetch raises a wait signal before do any operation on the process' memory).

Memfetch was design to target any process on a Linux environment then there should be any impediment to use it on an Android devices if its code is correctly compiled for the target hardware architecture.

But also with a method to read the HEAP there still sandbox environment which prevent any process to read any information belonging to another virtual machine. But again, as we show in the previous chapter, it is possible to bypass this limitation by obtaining root permission (by rooting the phone or by using an exploit).

4.1 Malware Implementation

This section explains in detail how our proof of concept malware was developed in order to first access, then dump and finally parse the targeted areas of the Android memory in order to retrieve user's personal information that can be later exploited for a specific user attack.

The application we developed is a working malware, except for the "spreading" part, which has not been implemented since it is out of the scope of this research. Specifically, the malware is a trojan application that transparently to the user steals personal information, such as login credentials, while on the foreground performs some visible useful tasks. This way, the user is deceived and does not notice the true nature of the malware.

The core part of the malware handles the memory dump. This procedure gives the capability to the malware to read and save information that is present into the memory area of any given process. Our function is capable of reading the memory sectors that the operating system allocates for an application, e.g. "Dropbox" or "LinkedIn" by searching for the respective processes "com.dropbox.android" and "com.linkedin.android" and then finding the information that it was intended to look for.

We based the dump routine on the existing memfetch code, specifically customized by us, whose purpose is to dump the memory of a program without disrupting its operation and it is normally used to examine suspicious or misbehaving processes at runtime. It is written in C language and it is designed to be compiled in a Linux environment. It uses the Linux function *ptrace* in combination with the process information available in `/proc/PID`. For more information on ptrace functionality. Android is Linux based and therefore uses similar processes and memory management. To complete the porting of Memfetch to Android, we compiled the source code in an Android environment with an ARM CPU design. More specifically, we used C4droid [26], a software available on Google Play. It is an offline C/C++ IDE and compiler for Android that supports devices with ARM proces-

sors and uses GCC to generate binary files. Instead of cross-compiling the code on a desktop environment, we used C4droid to compile the code directly on the Android device. The original source code was slightly adapted to the Android pages dimension and system libraries as follows:

- We removed the library *asm/page.h* because this library does not exist on Android environment. This change does not affect the proper functioning of the program.
- We modified the dumps final path to the *"/data/bin"* directory.
- We changed the *PAGE_SIZE* value to 4096 to adapt better the code to the target phone page size.
- We configured the code to save both *mem* and *map* areas present under */proc/PID* of the target process. It is possible to configure to dump only one of them by setting the respective flags to false.

The customized binary, named *savemem*, takes as input the process identifier (PID) of the application to dump. Since the Operating System assigns this value at runtime when the process is executed and it is not possible to determine it *a priori*, the trojan application browses at regular intervals the list of running processes using the *ps* command. Once the target process has been identified, it takes the corresponding PID value and feeds it as input to *savemem*. As result, *savemem* dumps and saves on files the content of the memory areas desired, in our case we are interested in the process HEAP and in the Dalvik-HEAP. These memory segments contains the raw information from which we will steal the reserved information as the login name and the password.

To complete the "trojan" aspects of the Proof of Concept (PoC) application, we embedded the *savemem* binary within an RSS reader application. An utility where the user can receive timely updates from websites he usually follows, without having to visit directly every single page to check for updates. We call this application *NewsShower*. To implement the application we used as development environment the Android SDK [23] and its IDE.

After setting up the environment, we configured Eclipse to use Nexus' latest Android API, which at the time of our research corresponds to API 18 (Android 4.3 Jelly Bean [24]). As testing environment we used a Samsung Galaxy Nexus i9250 with the same operating system version.

The *NewsShower* application is clearly a "dumb container", used for demonstrative purposes since it could be embedded within any application.

It is interesting to note two things:

1. We used, on purpose, a thread to implement the malicious part of the code for the following reason: the user will probably send the application in background at some point to use other applications, but wanting our trojan to continue checking also in background, with a separate thread and the use of a service, the trojan will be able to continue in background without affecting the normal device utilization.

2. The parsing routine merely checks for ASCII and/or Unicode strings present in the keyword lists. Although every application will keep this information in a different memory from the others, in the case of a targeted attack against a specific application, is very likely that the attacker will know exactly which keywords to search for.

To summarize, the steps of our PoC scenario are:

- When launched, the application checks if there is a copy of the binary file *savemem* in the `/data/bin` path. If the file is not present, the application copies *savemem* in the `/data/bin` path. If the file is already present from a previous execution/infection, then it does nothing and goes further.
- When the application is launched, it shows to the user all the feeds it receives from the sites the user has selected to follow. This step differs only at the first execution, when the program asks the user to enter or select from a set of predefined sites the ones he wants to select.
- In the mean time, a new separate thread that will handle the "trojan part" of the application starts in the background. The new thread will check at regular time intervals (e.g. every 20 seconds) if the target process is present in the list of running processes. The thread will keep checking in an infinite loop until the target process is found. We used a thread for a simple reason: the user will probably send the application in background at some point to use other applications, but we want our trojan to continue checking also in background. With a separate thread and the use of a service, the trojan will be able to continue in background without affecting the normal device utilization.
- When the target process is running, the trojan will grab the PID of the target application we want to dump. To do that, we used the class *ActivityManager* that permits to obtain all the running processes and then choose a specific process and its information (e.g. the PID) from the list.
- Once the PID related to the process we want to dump the memory from is found, the trojan executes the *savemem* binary. This task has to be executed with root permissions.
- After the dump has been completed, the trojan parses the resulting files looking for login credentials keywords such as *username*, *password*, *PIN*, *OTP*, etc., and saves them in a text file. The parsing routine merely checks for ASCII and/or Unicode strings present in the keyword lists. Although every application will keep this information in a different memory from the others, in the case of a targeted attack against a specific application, is very likely that the attacker will know exactly which keywords to search for.
- The elimination of all the dumps is the last step performed. After the personal information have been found, the dumps are no longer needed and can be deleted.

Every time we had to execute a shell command within a Java Android application, we used the Java command `Runtime.getRuntime().exec("command to be executed")` that executes the string specified between the double quotes in a shell environment. The commands executed by our application are the following:

```
(1) Runtime.getRuntime().exec("su -c chmod 777 " + savemem);
(2) Runtime.getRuntime().exec("su -c ." + savemem + " "
    + pid);
```

The first command is issued in order to assign full rights to the *savemem* binary after it has been copied on the device by the trojan, the second command executes the binary itself with root permissions and passes the PID value of the target process as parameter.

As described before, the parser simply scans through the raw data dumped from the memory and identifies the personal information. For example in the eBay application dump, login credentials are preceded by the strings "credentials", "userId" and "password", making it easier to identify them. The parser in fact searches for these strings and saves in a file the characters that follow.

To better understand, what follow is an example of one of the recurrent data in which is possible find personal data in an application HEAP:

```
6F 73 3D 61 6E 64 72 6F 69 64 26 61 63 63 6F 75  os=android&accou
6E 74 4E 61 6D 65 3D 31 30 32 39 33 38 34 37 26  ntName=10293847&
70 61 73 73 77 6F 72 64 3D 30 37 34 31 33 35 37  password=0741357
```

In this case there also the two specific labels "accountName" and "password" to facilitate the parsing phase.

At the end of its life, in order to leave as less traces as possible, the trojan removes the binary and the files containing the memory dumps invoking the *rm* command from the shell:

```
Runtime.getRuntime().exec("rm m*");
Runtime.getRuntime().exec("rm savemem*");
```

Finally, the stolen user information are stored in a text file and could be sent via SMS, Bluetooth, email or any other communication channel available in the phone.

Considering that a RSS aggregator has the internet permission to retrieve data from online websites, it is presumable that the stolen personal data will be sent to a remote server.

4.2 Malware Impact on System Performance

This section analyses what impact our hidden malware has on the mobile device. It is important to understand if the operations performed by our application, as described in the previous section, affect the overall performances and the user experience in a noticeable way.

In order to do so, we decided to monitor and benchmark the performance of the background loop which is searching for the target processes to appears in the running processes list, as this procedure has the biggest cost in terms of CPU utilization. The memory dump - the other main activity of the malware - is a single process that only lasts for a few seconds and does not affect in a noticeable way the mobile device. Therefore, we decided to monitor how the system performance are influenced by varying the malware background service scan frequency.

To benchmark the performance of the Android device, we used the AnTuTu Benchmark version 4.1.7 [46]. AnTuTu can perform different types of benchmark such as CPU and Memory, 2D Graphics, 3D Graphics, Database IO and SD card IO. We only ran the "CPU and memory" test, since these are the components that will be affected by our trojan application.

In order to better understand how much the frequency of the loop will affect the system, we ran tests in the following six conditions:

- Default: no third party applications are running. Only Android system processes are present.
- Loop checking for running processes every 1s;
- Loop checking for running processes every 5s;
- Loop checking for running processes every 10s;
- Loop checking for running processes every 30s;
- Loop checking for running processes every 60s.

The figures below show the test results: higher score indicates better performance. As it is clear from Fig. 4.1, the default system scored 3728 points and the 1s, 5s, 10s, 30s and 60s scored 3658, 3704, 3779, 3746 and 3723 respectively. Moreover, Fig. 4.2 provides the details of the test "CPU and Memory", which is the results of testing the RAM, CPU-Integer and CPU-Float calculations.

It was somehow predictable that system performance would be affected by the loop that searches for the target running process inside the malware application. Nonetheless such difference has a small impact on the system to be considered significant. Therefore, we may conclude that the simple step to check the running processes on a regular base does not affect in a tangible way the overall performance of the system.

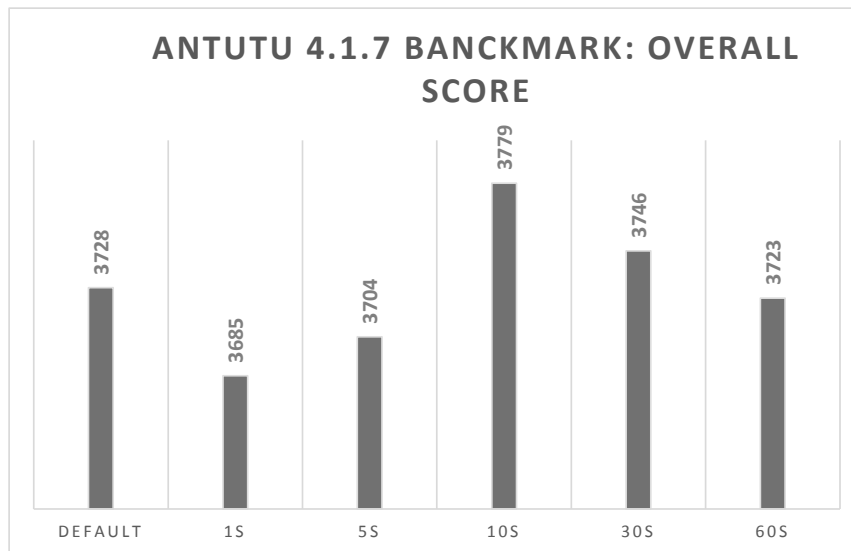


Figure 4.1: Overall benchmark scores from AnTuTu

The last test calculates the time needed to parse the HEAP dump file. The procedure scans the entire dump to retrieve the desired information. Every application stores this information in a different way, using however specific tags that make this procedure easier. As a result, the parser follows a different approach depending on

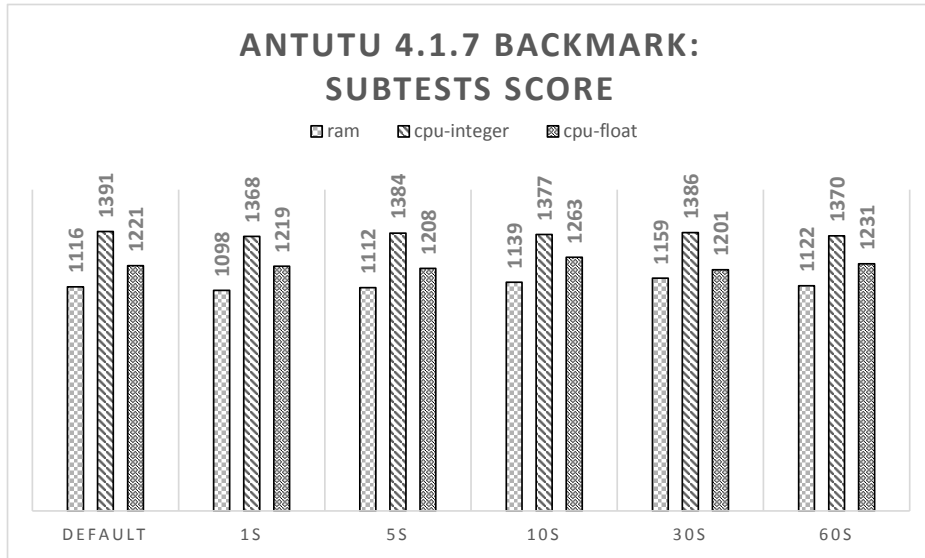


Figure 4.2: Detailed benchmark scores from AnTuTu

the target application: it contains specific patterns for each application we used in our tests. During the tests we measured the milliseconds the malware needs to find the information and to save them in a predefined location (in this case the SD card). Fig. 4.3 shows the times for four well studied applications: Twitter, Facebook, eBay and a bank application (the name of the bank is deliberately not revealed).

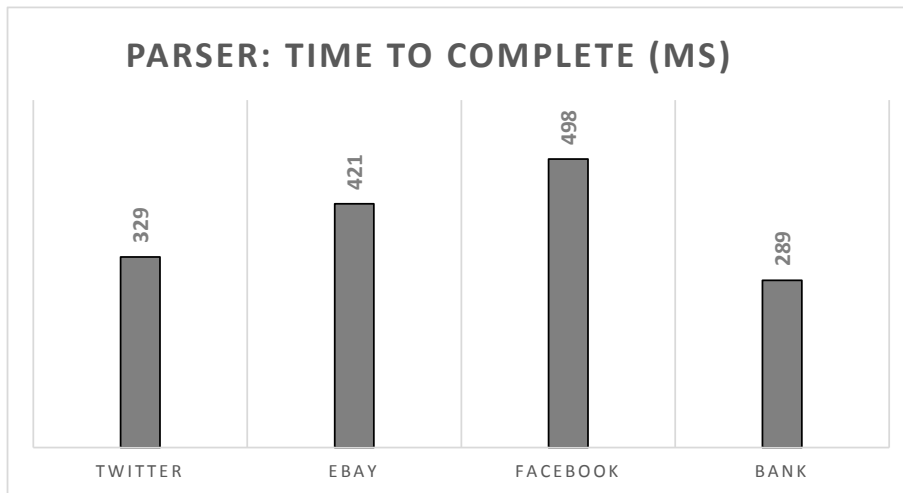


Figure 4.3: Time to retrieve and save the user information from the dump

Chapter 5

The ARAM Project: The Architecture

We begin to study security problems and focused on Android devices because today phones are not simply devices used to make and receive phone calls. Indeed, we call them Smartphone, since they have evolved a lot and have become much more in time. These devices are probably the most powerful collectors of sensible data for everyone of us. These facts make the study of security systems and personal data protection as fundamental aspects of the Android ecosystem.

Every day on internet we can read about some new devastating applications which stole information from our smartphones, use it for making money [2] and so on. We choose Android because it is the most widely used operating system in the world for mobile devices and its open source nature. Obviously this makes it also one of the favourite target for crackers and thieves from all the world. Besides the need of security in this context there are few tools that guarantee an acceptable level of security. From these, the need to create a new application that can identify a menace on such an important device.

Literature, as we saw in the previous chapters, is rich of works targeting Android security problems. While every study proposes a different solution to face malwares and generally any malicious activities, all of them have something in common: they require to modify the Android underlying structure or to have full access to the phone (root permission). These are really strong requirements. First of all, a reasonable solution should be able to reach every device on the market. Users don't know what a firmware, a kernel or a libraries are, they want just to use an application to get a benefit. A non technician would be never able to modify the kernel structure or to flash a new firmware which integrate anti-malwares code. Also the root requirement is too strong and not available to the mass directly. Then to reach any device the application must be installable without requiring any modification or specific skill. The solution should be distributed as any other application and require just the pressure of a button to be installed and used. Second it should not alter in any way the user experience or if a change is really indispensable it must be the lesser possible. User should no notice any change in the way he uses its phone. Any interaction between the users and the "solution" should be avoided or minimized to specific events such as a compromise system.

For these, and many other reasons in the following, we develop the ARAM frame-

work, a complete architecture to control in time the behaviour of an Android mobile device.

Due the strong restriction we imposed above, we have to analyse the behaviour of the device and the respective installed applications at higher level possible. In fact, we do not know when applications call system APIs or other functions which need system resources without modify the kernel. However, we do not want to do this, because we want to realize a system which can be used on Android, as it is out of the box.

We decide to perform our analysis not on the structure of the source code nor on the compiled APK file but on the profiling of all the applications and the behaviour of the device itself. We need to analyse what the applications installed on the device do in order to identify malicious behaviours.

We have not performed an analysis of the source code because we want a software which is capable to perform an analysis in time. Also in an Android application is possible to insert a pre-compiled C code, which makes this kind of analysis inefficient. Anyway how will see in the Application Module, we do also static analysis on the information available from the manifest file but with a novel approach thanks to the centralized server.

ARAM is a modular project and its function can be adjusted according to the user needs. In the following we will see first the overall idea and architecture and then we will analyse and see the contribute of every module composing the framework.

5.0.1 The Main Idea

ARAM, acronym of Android Real-time Analyser Monitor, is a framework designed to analyse and to control the behaviour of mobile devices having Android as their operating system. ARAM has a client-server architecture as shown in Fig. 5.1

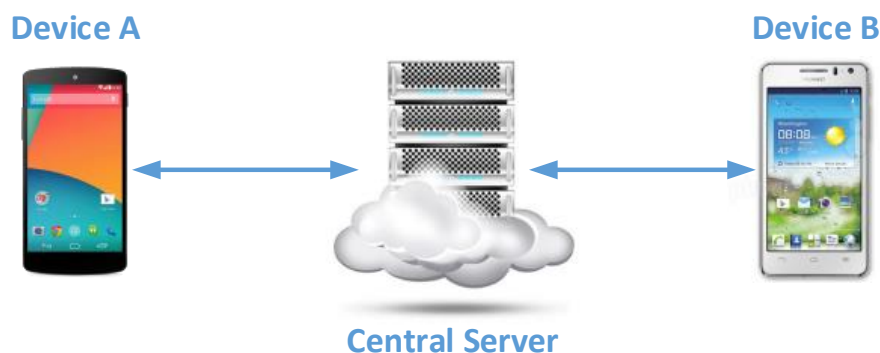


Figure 5.1: ARAM overall architecture

It is composed by a mobile application installed directly on the mobile device and a centralized server. The main idea is quite simple: collect a series of information directly from the mobile device and send them to the remote server in order to analyse the data and understand if the device behaviour is normal or not. It monitors most of the activities happening on the phone to keeps a constant report of the state of the device and creates snapshots of its current activities. This way, we can create a profile of the applications installed on the phone and try to identify the presence

of an infection. However, we need to collect as many states as possible to understand what is happening and if there is any malware running. As we already said ARAM is a modular framework and what could consider normal or abnormal behaviour depends on which module is used to analyse the data. In fact each module has its own specific function and works to accomplish a complete different task.

Totally there are four distinct modules:

1. Snapshot module: it permits to collect the device information and to analyse them manually on server side;
2. Policy module: it permits to express behavioural rules and to control them over the time;
3. Application module: it permits to control the application installed by the user on the device and discover tampered versions;
4. Identity module: it permits to enforce the user identity while performing an online action on a third-party web site.

Each module is divided in two distinct parts: one on the mobile device and one on the server side. Each module communicate over a common communicational channel as show in Fig. 5.2. All the modules have one characteristic in common: they do not perform any analysis directly on the device but they transfer the needed data to the server and wait for a response. It is the respective module on the server that will perform any computation.

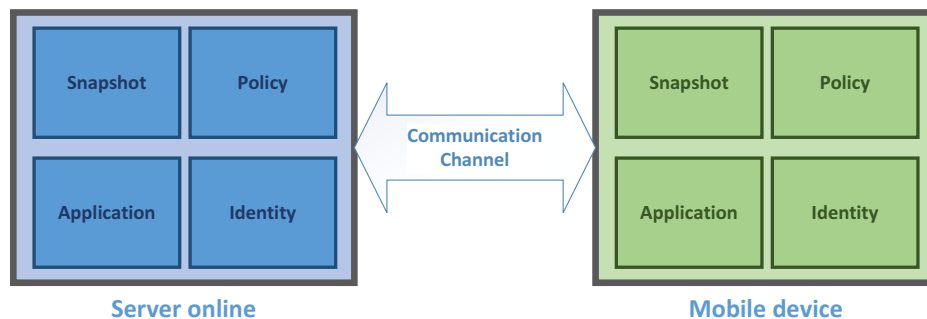


Figure 5.2: Four module communication

One of the advantages of our framework is the possibility to confront data from different users and mobile devices to improve the discovery of malicious activities. This could be done through the centralization of the data on a server. This permit to us to trace the behaviour of applications used by registered users and also to identify if an application works in suspicious way on some devices and not in other. Further, this permit to alert an user even before an infected application begin to act in a harmful way.

Without the centralization of data, it is not possible to interweave the behaviours of the applications and identify malicious actions. In general, every module benefits from the use of centralized data. We will further investigate these advantages in the respective sections. Also, we decided to send these data to an external computer because, nowadays there are no phones that can make complete analysis directly on

the device itself without a consistent loss in performances and without a massive battery drain. As we said, we don't want to affect at all the normal user experience.

The collected data in this specific module could be used for data mining and statistical analysis on server side but also to monitor and then profile the behaviour of the applications installed on a phone. All with the scope of understand if a certain application contains malicious code (a malware) or its behaviour is rightful.

While the statistical part could be useful for many purposes as commercial, promotional and even parent control, our interest is related to the malwares and illicit behaviour discovery.

However, understanding what an application is doing and if it is or contains a malware, requires to collect a great amount of data. For example we need to know when the screen is ON or OFF, how much resources are used both from the system and from the APK installed, how much data traffic each application is exchanging over the available communication channels, and so on. Moreover, we have to know all this information at the same time, at every prefixed interval of time. Without these data we are unable to correctly profile the app installed, and consequently we are unable to identify the presence of an anomaly.

We selected several information and sensors we want to keep an eye on before we can start to do our research. Therefore, we perform an a priori selection without knowing which of the selected "objects" would be significant for our scope.

We believe that if we store and analyse in time a great amount of states of the device, we are then able to identify a new malicious behaviour on that phone. Obviously, this means that if a malware is already present on the phone before our application is installed, we can not guarantee its identification. However, if we collect data from a great number of devices we may be able to detect the presence of an infection thanks to the comparison of different profiles. Cross different data permits further level of data mining.

Our approach requires some assumptions:

1. there is the possibility to not identify a malware due to the fact that its influence on the system is negligible. This could happen if the changes it brings to the system are so minimal that could be considered a normal fluctuation in the registered parameters. In these cases the comparison of data coming from different users could improve the discriminatory capacity because the same app could bring more pronounced changes in other devices.
2. the initial gathering of data should be done on an uninfected phone. This is a fundamental constraint if we want to maximize the capacity to compare two different behaviours in time, probably with a different set of apps in execution.
3. a continuous gathering of data causes a little loss in performances but this is amply justified by the advantage and security countermeasures we introduce. We must always remember that the damage introduced by a malware could be catastrophic.
4. user must have a data contract with his operator if he wants to use ARAM constantly.

5. the real-time information analysis requires a constant active internet connection between the client and the server. If this connection could not be guaranteed by the provider than our architecture should be considered as a near real-time system.

The first point is absolutely true whenever we talk about security system applied to whatever devices. In our case we have to say that our study is based on the observation of the applications behaviour. Then, if a device modifies its own behaviour we are able to discover it. The second point of our list is also important. We need to have some initial data to be considered uninfected to fix the thresholds correctly. If an application is already infected when we began our analysis, the identifications will be harder to do. However, there is the possibility to identify those applications by comparing the behaviours between all phones. In fact, in many cases, some applications work in similar way regardless of the device and the users who runs it. Point three is a directly consequence of the fact that we need to monitor the states of the phone at fixed intervals. For the performance loss, Android manages autonomously the execution of the active applications and our application could be killed at any time to save and free resources. We need then to bypass this imposition and have a service not killable and "freezable" by the Android operating system. Obtaining this requires to force the constant presence of our client in the CPU process priority list and then have a decline in CPU rest. We will go further on the performance analysis in the Snapshot chapter. Point four is a weak constraint because most of the users own a data contract with their mobile operator. Further our architecture works also with a normal Wi-Fi connection. Point five is easily explainable: the internet connection is the only part of the architecture we can not control. To guarantee a real-time system we may fix an upper bound in the server response time but if the device could not be reached then such bound is useless. Following the narrow definition of "real-time", we may consider ARAM as a soft real-time system. If a response deadline is defined and, due to the internet connection, it could not be achieved then the user will be alerted with a not predictable delay. The consequence is not a total system failure but thereby a degrading in the quality of service.

In the next chapters we will see one by one all the ARAM modules.

Chapter 6

ARAM: Snapshot Module

The Snapshot module is the base of all the ARAM architecture and it is composed by two distinct parts. The first one is a mobile application which is responsible to collect the data of the device where it is running and then to send those data to the ARAM server. The second part is a cloud structure composed of a server where all the data from the users are collected.

The module scheme is shown in Fig. 6.1.

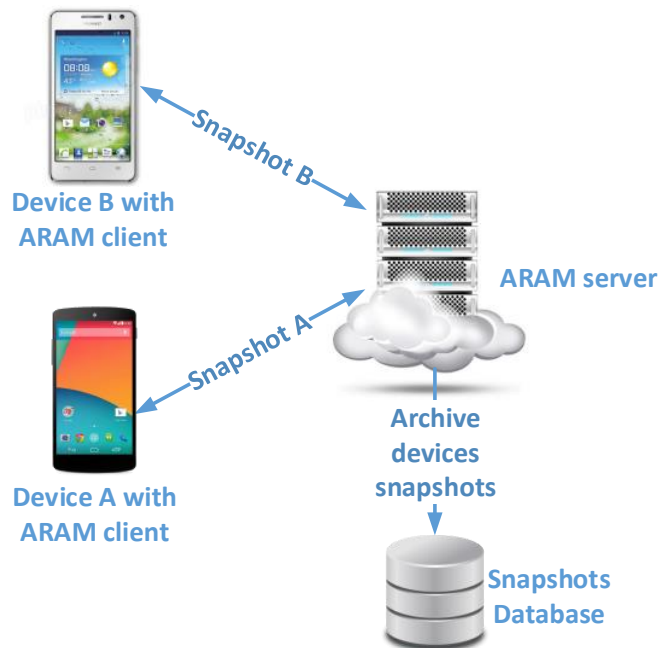


Figure 6.1: The Snapshot Module scheme

ARAM client, for the Snapshot Module, has a really simple interface and a limited set of functions, transparent for the user, which are used to take a *snapshot* of the device. Indeed, a snapshot is a "picture" of the states of each relevant sensor present on the device and indicates what the phone was doing at that precise instant. The collection of these snapshot every 15, 30, 60 or 300 seconds can give us a description of the behaviour the phone had in a precise period of time. The

more frequent are the snapshots, more precise is the profiling of the applications. Analysing this behaviour let us identify an unwanted use of the phone.

Therefore, ARAM is an application which collects data about how the installed applications use the resources and, more generally, about the habits of the user. ARAM client itself is not an application that increments the level of security of the device but allows other software to do a deep analysis and to identify a potential menace.

As we say nowadays it is not possible to offer a runtime protection on a smartphone without losing an incredible amount of performances. Hence, ARAM sends all the data it collects to an external server. This computer receives the data from all the phones that have installed the client. Using their IMEI codes, it is able to identify them and compare the new data with those previously sent. The architecture of the server will be shown later, for now we focus on the structure of the client.

6.0.2 ARAM's Client

ARAM's client store the data in the phone where it is running. More precisely, we store our data in a SQLite database. We choose this database format because it is simple, compact and fully integrated in Android. The database structure is really simple because it is composed of only one table. We did this on purpose to keep the device part as simple as possible. Each row in this table is a snapshot of the phone and represents the value of each sensor we choose in a specific instant of time. This is the fundamental core of the application: Android is not made to collect all the information simultaneously, we use a row as a buffer until all the required information are not available. Without this, we could not write an application that can create a "immediate" snapshot of the phone. Together with the sensors, we collected other useful information thanks to the Android API.

Let see in detail which information are acquired and what they represent in the overall profiling process.

Column Name	Description	Data Type	Accepted Value
_id	Index for each row	Integer	1 to infinite
_cell_id	Unique code of the phone such as IMEI	Integer	15 digits Integer
Screen	Point out if the screen is ON or OFF	Binary	0=OFF 1=ON
CPU	Global CPU usage	Integer	0 to 100 (%)
RAM	Global RAM usage	Double	0 to 100 (%)
Process	Running processes information	mixed	<app_name: cpu(%): ram(%): owner>
Disk	Free memory indicator	Integer	<Internal; External>
Proximity	Distance from the phone	Double	positive decimal value
Light	Ambient light illumination	Double	

Accelerometer	x,y,z axis acceleration of the phone	mixed	<x;y;z>
Battery_Level	Percentage of battery left	Integer	0 - 100
Battery_Power supply	Type of Power supply of the phone	Ternary	0 or 1 or 2
Battery_Temperature	Temperature of the battery	Double	
Network	Currently active default network	ASCII	
Data_Received	Total number of bytes received from mobile network and all bytes received by the phone	mixed	<mobile;all>
Data_Received_For_App	Bytes received per app from all available networks	mixed	<app_name: traffic>; <app_name: traffic> ...
Data_Transmitted	Total number of bytes transmitted from mobile network and all bytes transmitted by the phone	mixed	<mobile;all>
Data_Transmitted_For_App	Bytes transmitted per app from all available networks	mixed	<app_name: traffic>; <app_name: traffic> ...
Calls	Number of calls made, received and missed	mixed	<made; received; missed>
SMS	Number of short text message sent and received	mixed	<sent; received>
Steps	Steps done	Integer	
GPS	Current location of the device	mixed	<latitude; longitude>
Timestamp	Instant of time the snapshot has been acquired	Integer	epoch time

Table 6.1: ARAM SQLite DB

While the database has always the above fields, some devices doesn't implements all the sensors we want to record. In these cases the sensor field is filled with a special string and the server skip to perform any control over it to save resources.

Let see the list in detail:

_id and _cell_id: The first value is a simple id of the row of the table, while _cell_id indicates an unique code of the phone that sends the data to the ARAM server. This value is used by it to group all the data received by a single phone;

CPU RAM and Screen: We obviously consider the global use of the principal resources such as CPU and RAM, and we add the value of the state ON/OFF of the Screen, as a binary value. This is extremely important for our analysis because Android try to minimize, when possible, the use of resources if the screen is OFF. Android does this to save energy and battery life. We expect high percentage values in the CPU and RAM usage when the screen is ON, but not when the screen is OFF. On the other hand, some malwares use CPU intensively only when the screen is OFF, so that the user cannot see the loss in the efficiency on the device;

Process: This column contains a list of the processes running on the phone. This field is mixed and contains: the name of the process, the percentage of CPU and RAM used by it, and the owner of the process. With all this information we can try to identify which application is in background and which one is effectively running at each instant of time. This may help us to identify which applications are more frequently used and how many resources they use, for example to identify an always running malware. In addition to this, these values creates the basis for the application profiling;

Disk: We collect information also about the mass memory storage of the device: in particular we insert in the Disk column the value of internal memory and the value of an eventual external memory, such as an SD card;

Proximity Light and Accelerometer: These three columns contain values about the distance from the phone (the first object in line with the sensor until 5cm), the ambient light, and the values of the accelerometer. Those values are not strictly related to our scope but they can be useful to better understand what is happening on the phone;

Battery level, power type and temperature: These contains information about the current level of battery, the type of power supply and the battery temperature. These values may indicate how much global resources are used. Not only CPU and RAM but also mobile networks, data traffic, calls and text messages. In fact, those resources consume battery power, and greater is their usage, greater will be the battery power consumption. These information may be useful to indicate if there are resources used abnormally. We register the type of power supply because some malware activate their malicious code only when the phone is connected to the power supply. Other malware, instead, try to infect personal computer when the Android phone is connected to the PC. Keeping track of this value may help us in identifying these behaviours.

Network: We reserve a column to the information about the active network connection. This field is a text value and indicates the default network used by the phone. This means that the phone can have more than one connection active, but we insert in the database only the one that manages data traffic in that instant;

Data Traffic: After network connection we can find four columns containing all the data about the amount of traffic received and sent. In the first column of these four we register the global data received in bytes. The field is multivalued

and represents respectively the received data by mobile network and all the available data received among all the networks. These two values are the same in case of no other network is used except the mobile network. The second column of this group contains all the traffic received for each application. This is a multivalued field containing all the applications that can receive data from network and the amount of byte received. The following two columns are exactly the same as above but the values represents the traffic sent not the one received. The third column contains the quantity of bytes sent by mobile network, following by the bytes sent from all the networks available on the phone. The last of these columns contains the bytes sent by each application in a specific instant;

Calls: This column contains three values, respectively: outgoing, incoming and missed calls. We want to monitor these values because there are malware that can make calls in a completely transparent way for the user. An attacker can make a phone calls a premium rate telephone number without the user noticing;

SMS: Same as above, we track the amount of text message traffic on the device. The column SMS indicates two values that represent respectively the number of SMS sent and the SMS received;

Steps: This column contains the step done by the user from the midnight. Every day the field is reset;

GPS: This column contains information about the position of the device. We collect latitude and longitude because there are malware that running only if the device is in a particular geographic area. Furthermore these values permits us to geolocalize an infection and may tell us more about the attacker. However, despite its name, this information is obtained by the device operator. We choose to use this instead the GPS signal because, this way we can obtain information about the position of the device even when it is inside a building. Even if this information is a little bit imprecise rather than the GPS, it does not consume further battery power. Also, the GPS sensor cannot be activated by an application without the user consent and we do not want to prompt a request to activate it. We want ARAM running in a completely transparent way, until something suspicious happens;

Timestamp: The last table's value is surely the most important. This field contains the information about the time we collected all the above data. Without this value we are absolutely unable to track the behaviour of the device, because we cannot tell if a big variation in two following row is caused by an application or if it is caused by a big interval of time between those.

Table has no other columns, and from now on when we refer to a snapshot we pointed out a row of this table composed by all these values, as shown.

ARAM mobile application is built to be a completely transparent application until something suspicious happens. This means that until the last snapshot sent do not exceeds a threshold value in the analysis, ARAM server do not consider the

behaviour as suspicious. In the opposite case, ARAM server will send a message containing the details of the suspicious activity. Then, ARAM mobile application prompt it to the user. Snapshot Module doesn't directly have automatic analysis routines and do not offer any capability of discriminate between behaviour. These tasks are offered by the Policies and Application Modules which we will cover in the next chapters. Snapshot module just collects the information and permit to analyse them statically through a specific web interface.

The client has a really simple user interface. When the application is executed for the first time, after the fresh installation on the device, it is shown the Login or Registration interface as illustrated in Fig.6.2.

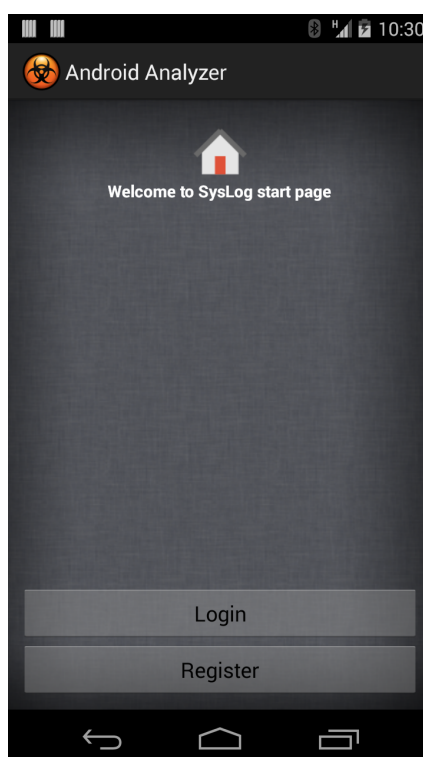


Figure 6.2: ARAM's client

In this first page the user can invoke the Login page if he is already registered to the service - from a previous installation or by using the web registration form - or the Registration page if he has never joined the ARAM service before. The application keep trace of any previous installation and login process by storing a file on the system. If the user has already logged then the interface is not shown at all. Fig.6.3 shown the registration form. The page asks to the user to insert his personal information.

- Name: user's real name;
- Surname: user's real surname;
- Username: an unique name the user could choose to represent his person;

- Password: an ASCII string;
- Email Address: the user's email address.
- Phone Number: user's phone number;

Username should be unique because, together with IMEI number, are the information used by the system to discriminate between registered users. IMEI is collected automatically from Android API because it is an information too technical to be known.

The Username and password combination could be used to login to the server if for some reasons the device has been formatted or the application removed. They can also be used to log in the web site to see statistic and use the analyser tool (we will see it later in the chapter).

Email and phone number are asked for the same reason. They permits to have alternative way to communicate with the user if the phone has been compromised by a malware. To have a second channel communication is fundamental. In the setting page the user could personalize his alert system and choose how to be alerted.

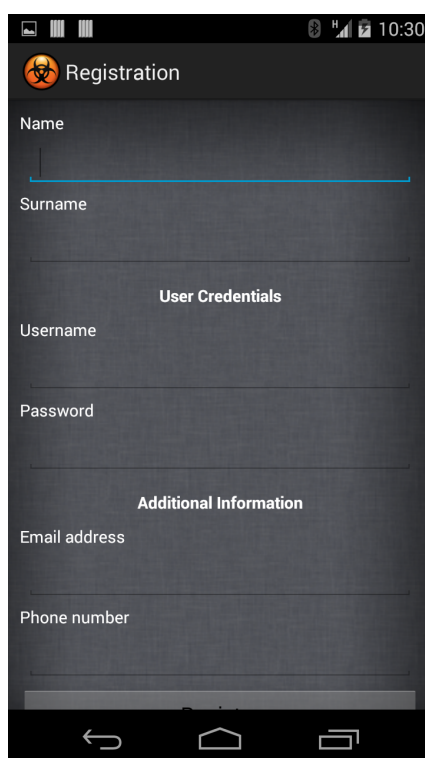


Figure 6.3: Client registration form

The Login page, not show there, just ask for the Username and Password information. After the Login process has been completed the user is bring to the activity main interface of Fig. 6.4.

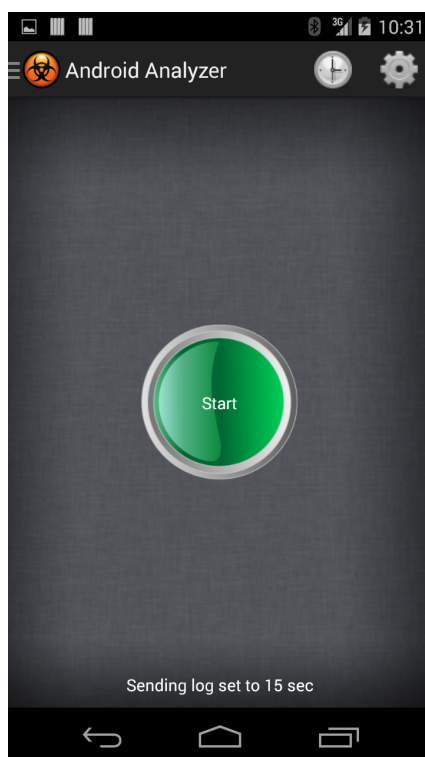


Figure 6.4: Client general interface

Main activity present only one central button and a top bar to change system parameters and invoke other ARAM modules. The button is used to start and stop the service. ARAM collects data only when the service is running. The user has only to tap this button to start the recording process.

On the left side, the top three grey lines permit to load the other functions offered by the ARAM framework as the Policy Module and Application Module. On the right top side are available two icons. The clock icon permits to select the frequency of the service, aka how many time intercourse between two different snapshots. This is an important step to analyse. The recording process is handled by a single thread. This is done for resources constraints. Our service is running 24/24H and with modern smartphone with two or even four distinct CPU cores it is possible to allocate one of them to the service and have a very low impact on the user experience. Also this is due to recording purpose. Android work with events and the only way to record correctly all the data requested by a single snapshot is to perform the acquisition in sequence. For these two reasons the frequency between snapshots is to be understood has the time passing between a snapshot end and the begin of the following. This doesn't influence at all our approach and the delay could be considered irrelevant. The user can select between various frequencies: 15 seconds, 30 seconds, 1 minute and 5 minutes. The frequency influence both the device performance and the analysis quality. The performance because, to perform a single snapshot, all the sensors and resources must be interrogated with a peak in CPU and power consumption. Also the data has to be send to the remote server and the use of a cellular or wireless communication affect the battery duration. We will see later some graphs to understand how. The quality because to more snapshots

correspond more data to work on and then more information to extract. A large interval between two snapshots may not point out a malicious behaviour, so the ARAM works may be useless. More frequent are the snapshots, more precise the difference in the states are and it is easier to understand what is happening on the phone, both in foreground and in background.

All the data are collected and sent through the internet, or inserted in the database, in a way completely transparent to the user. The device owner see only how many states have been already registered as shown in Fig.6.5. This information is visible in the top drop-down menu of Android. This is the only information available and visible to the user while he is using the smartphone (and then other GUI are visualized). In the same place is eventually indicated the presence of an anomaly (and further information with a full description are available in the ARAM application GUI).

The application sends automatically the data if there is an active connection, otherwise it stores the data on the device memory and sends those when an internet connection is provided.

In detail the application works in this way:

1. The application acquires, from sensors and Android API, the information which composes a complete snapshots;
2. The snapshot is saved, on an internal folder device, as a tuple of a SQLite database. Each snapshot corresponds to a database line.
3. If an internet connection is available (over Wi-Fi or cellular network) then the client send a POST request over HTTPS to the server. The information sent are the IMEI value and a JSON Object [50] containing the whole snapshot as an ASCII string. JSON is a short for JavaScript Object Notation, and is a way to store information in an organized, easy-to-access manner. We will not cover the implementation in details because it is not the main purpose of this work the implementation aspects. If the server received the snapshot then return a positive answer to the client which delete the snapshot to save internal memory.
4. If the internet connection is not available then the acquired snapshots are saved in the internal SQLite database and sent in a whole block when the connection will be available.

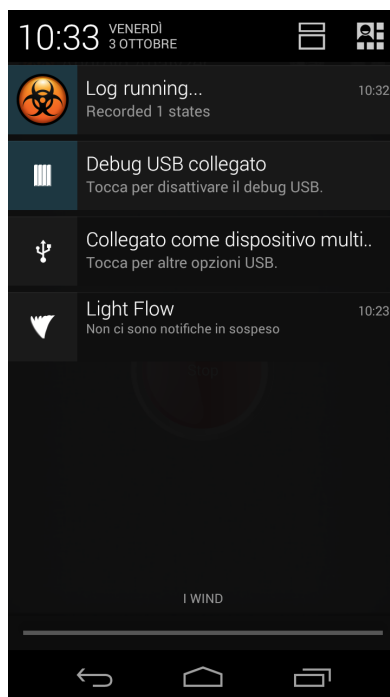


Figure 6.5: States counter

6.0.3 ARAM's server

Now we present the server side structure software of ARAM dedicated to store the information provided by the Snapshot Module.

The server software is built around the XAMPP distribution [35]. XAMPP offers a complete Apache distribution containing a MySQL database and a Tomcat implementation. This open source project permitted to us to concentrate our effort on the ARAM architecture and to the testing phase without wasting time to develop and integrate all the single software we needed to realize our ecosystem. We used directly MySQL to design and manage the database and HTML and JAVA for the interface and the computational parts.

The database of ARAM, server-side, contains many tables. Some of them are used directly by the Snapshots Module while other are filled by Policy and Application Modules. Here we will cover just Snapshot Module tables.

The first table is *Users* which contains a small set of values, exactly the ones required client side during the register procedure. To assure the uniqueness of the user-device pair we choose to use the IMEI (or MEID/ESM depending from the location).

Second table is *Snapshots* which records all the snapshots received. Every line is a single snapshot and it is possible to identify the owner by the IMIE contained in the fields.

Three further tables store same average values calculate in-time while the punctual snapshots are received. The three tables contains the same data structure - a subset of the sensors which means were significant - but is different the interval of time on which they are computed: hourly, daily, weekly and monthly intervals.

The subset is so composed:

Screen: field shows on how much states the phone was turned on;

RAM: the mean value of the memory usage in the interval;

CPU and RAM: the mean value of processor usage in the interval;

Process: The same happen with this field, but in this case we compute the mean values of CPU and RAM for each process present in the rows of the chosen interval;

Disk: This column contains two integer values, positive or negative, which indicate how much space was occupied, respectively on internal and external storage in the interval;

Battery Level: The same thing as in the Disk column is done with the Battery level. It record how much battery was charged or discharged using an integer, positive or negative, value in the interval.

Battery Power supply: In the field Battery power supply is recorded how much states in the interval the device was battery powered, USB powered and AC powered;

Battery Temperature: Battery temperature, instead, contains the mean value of the battery temperature, expressed in Celsius degree during the interval;

Data Traffic: The next four columns contain the amount of data sent and received, respectively `Data_Transmitted` and `Data_Received`. The other two column of this group contains the amount of data sent and received by each application, respectively `Data_Transmitted_For_App` and `Data_Received_For_App`;

Network: In ARAM server, Network field become a mixed value and it contains for each available network connection on the device, on how much states that network was the default connection;

Calls and SMS: These two columns contain the number of calls and SMS sent and received. As in the ARAM mobile database, those columns contains mixed values. Calls contains the number of calls made, received and lost. SMS instead contains the number of text messages sent and received.

The columns about the data traffic, the calls and the SMS are calculated by doing the difference between the last and the first snapshots in the selected interval. This way, we can identify how much increment there was in the interval on those data. Not all the averages are calculated in the same way. For field as the CPU and RAM, they are calculate as the sum of measured value contained in the snapshots divided by the number of snapshots. For field as the Traffic, calls and SMS the average is the number of bytes sent/received, number of calls done and SMS sent respectively in the interval chosen. The system calculated these average while the snapshots are received. Hour average is calculate from XX.00 to XX.59 where XX is the hour in European format (0-24). Daily average from 00.00 to 24.00 of the day and so on until the month. Another table records a series of information

related to the applications installed by the user - as application name, application permission, application signature, application hash, application version, maximum data transmitted and received for single app between two consecutive snapshots. These information are better described in Application Module chapter.

The last table collects the maximum value registered while calculating the average for hour, day, week and month. The fields are update only when, calculating an average for the previous tables, one of the value is greater than the one registered in the last table. This table contain only four tuples. The server interface is a simple web page in which users can log in a see their snapshots and mean values. Login name and password are the one chosen on the device while registering to the service. An user could see only information generated by his own device. Only the administrator account has the power and the ability to see and navigate between all the user information. Fig. 6.7 shows the website main page after the login by the administrator. The user interface is the same except for the possibility to choose between devices from the IMEI list on the left.

Normally the page shows only the last 50 snapshots but the user could navigate between his whole data if needed.

Instead Fig. 6.6 shows the daily average values. User can select the temporal average interval from the top menu.

Other parts of the interface are dedicated to the Policy Module, we will see them in the appropriate section.

6.0.4 Hardware

ARAM has been developed as a proof of concept. The hardware at our disposition was limited but enough to test the effectiveness of our method. To implement the server we used a desktop computer with an Intel E6600 (dual core processor) and 4GB of RAM. The network, server side, has been implemented with a 10/100 LAN connected to internet through a router device. On client side the communications were performed through a Wi-Fi connection over a domestic ADSL line while the cellular communications were performed through an UMTS connection.

6.0.5 ARAM's Client Benchmarks

Another important issues with our mobile application, is the power consumed. As we show in first chapter, Android manage the applications at OS level. It reserves the right to stop an application or a Service if there are another one with higher priority. This may causes an interference in ARAM's Service. For example, there is the possibility that the application is not able to collect the data we need on constants intervals. For this reason, we need to maintain our application always running. Obviously this causes a loss in power energy.

Our application starts a service in order to record data values. In this way we are able to collect data even when the screen is OFF and also, we can obtain those at precise intervals of time, as we need. However this causes a little loss in performances of the phone. The service never stop and the CPU never goes in power save mode.

The first we performed is to understand how the application affects the overall performance of the device.

To that scope we used Antutu Benchmark [46]. Antutu is a tool, freely available for the Android platform, which shows a global parameter and other details about the performances of the device which is running on. Like other benchmark apps, AnTuTu gives your device an overall numerical score as well as individual scores for each test it performs. The overall score is created by adding the results of each individual score together. These score numbers don't mean much on their own; they're just useful for comparing different devices. For example, if a device's score is 10000, another device with a score of 20000 is about twice as fast. The overall score is based on RAM, CPU, GPU and I/O performances. Normally the tests are executed without any application running (except for the one needed for work). In our test we will see how the execution of the ARAM client change the overall Antutu score.

In Fig.6.8 are shown the benchmark results computed on a Samsung Galaxy Nexus GT-9250 device with Android 4.3 installed. How is clearly visible, the overall system performance are lower as the acquisition frequency is closer. In detail the performance are 10% lower with a 10 seconds interval and just 1% lower with a 60 seconds interval. For hour purposes and to not infect the performance, a time interval of 30s or 60s is sufficient. Besides, many resources recorded are considered as the quantity of the specific phenomenon has happen since the last snapshot. The field as calls, SMS and data send/received are not influenced by the frequency of acquisition because they report how many calls, SMS and data has been sent or done since the last snapshot. The frequency is not important also for battery capacity and temperature. Neither for the screen because is improbable the user will turn on and off the screen so frequently to not catch the change between a short interval. The only value really influence by the time is the CPU but is likely also the one more influence by a process malicious activity in background.

The second test try to understand how the battery residual capacity is influenced by the ARAM client. Again the test is performed acquiring the snapshots at different intervals. Result are shown in Fig.6.9. The trend is very similar to the Antutu one. Higher is the acquisition frequency and higher is the energy computation (and then the battery lasts less). With a 10s interval acquisition the overall decay is about the 20% (the phone battery duration is 2 hours less). Considering the overall market situation with smartphones which don't last until evening with a single charge, 2 hours less could be considered a sever impediment for many users. The situation improves using higher frequencies acquisition. With 60s interval the duration lost is about 11% (a little more than an hour). This loss is acceptable if we consider the benefits from the use of the service. Moreover, the ARAM client may be further optimized and the situation could be improved. Anyway the internet traffic is the part requiring more energy and this part, if we want to remotely control the device, cannot be avoided.

The last test wants to measure how many bytes are transmitted to the server over 24 hours of continuous acquisition. The result are shown in Fig.6.10. This test is very important to understand if the service is serviceable also when a Wi-Fi connection is not available. Many smartphone contracts offer a limited quantity of traffic over internet for month. For example, in Italy is common to have just 1GB or 2GB of available traffic for month. Our tests indicated that with a 10s interval are generated 24MB of data per day and then 720MB monthly. If the user is usual frequent media

channels such as YouTube than combining his habits with ARAM service could be a problem and the monthly traffic not be enough. Again if we bring the interval acquisition to 60s the traffic generated are definitely lower. With an average of 180MB generated the service effect less the user habits. The traffic generated can be easily turned down leaning when it is possible to a Wi-Fi network.

6.1 Malwares Implementation and Analysis

In Android we have a great number and type of malware. According to Zhou [25] there are three kind of attacks to infect mobile users. The most frequently used are:

- repackaging
- update attack
- drive-by download

Always according to the same work, the first one is the most widely used technique in Android devices. It consists in identify a popular application, download it and add a malicious payload then re-upload the infected application to various Android marketplaces. This kind of attack can easily affect users who want to install premium application without paying these. An attacker insert a malicious code inside a popular premium application and then offers it for free on unofficial markets. Users who do not want to pay for that application, install it from unofficial store and they got a malware.

Update attack does not directly infect the target application. Malicious code is being presented as the update of that application and, also in this case, the user voluntary download it. Instead, the third type of attack is similar to web based attacks. Those try to secretly redirect the user to a malicious download page through aggressive advertising in app or infected QR code. However some malware can use simultaneously more than one of those attacks. According to [25] approximately the 86% of unique malware they have identified in 2011, on a sample of 1260, were been created using repackaging. This technique is quite simple to create Android malware. First of all, we do not need to create an awesome application to get users download it. We need only to understand what this application do and how it works. Then we can insert a new service, or use one already present, or simply inject some code in the *onCreate()* method, see Fig.2.2. If the application we choose does not require a permission needed for our malware, we add it in the application's Manifest file. However sometimes this can be a problem. An expert user could notice that an application require more privileges than needed and decide to not download the application. However, according to Berkeley's university study [21] only the 17% of users pay attention to permissions message. Even worse only the 3% of them understand the meaning of permissions. Seem that users are generally more attracted to what an application says to do, instead of what it really does.

Now we show the malware we create to collect data for our analysis. We create those using the repackaging method. We did not use real popular applications, but instead we works directly with open source applications downloaded by F-Droid [37]. We did this because we did not want to disassemble popular APK files, because it would have been a loss of time for our scope. Also, we do not redistribute our

malware, but we use those only for research purposes. Below we present some of our malware and explain their functioning.

It is important to stress the fact that we use the mechanism of permissions to achieve our malicious objectives. However, it is possible to avoid these if an attacker uses an Android or Linux bug to obtain root privileges. In this case it is not important which permission an application requires, because with root permissions there is nothing that an application can do.

6.1.1 Hypnotoad

Hypnotoad is a live wallpaper coming from the world of cartoons, Futurama. We download it from the F-Droid and we inject in it malicious code using repackaging technique. This one is an example of experienced user recognizable malware. In fact, we choose to add a permission to the applications which consent a completely access to the networks, so we can send data through internet. If the user reads and understand the permission list, he can find that Hypnotoad requests *Network Communication* permission. Now he can figure out that maybe there is something wrong. The user has to wonder why a standalone wallpaper needs this permission. So if the user knows this things he can avoid some menaces. However as we already said, users are generally unaware of what a permission is or what it means. In addition to this, unfortunately, most of the users do not even read the list of the application permissions.

Once installed, Hypnotoad reads installed applications' list and running process every time the wallpaper is visible. Then every time the user look at his/hers Android homepage, Hypnotoad sends a mail to the attacker containing those information. An attacker can use this data to collect information about popular applications, exploit known weakness about some installed applications and to profiling habit of the infected user. All this using only the internet permission and completely without the user notice. In fact the collection of those information do not cause any visible loss in performances and the sending of the mail is completely transparent for the user. But Android has inserted a security system to block the sending of mail without the user's direct consent for each e-mail. We find a method to partially bypass this security level. For our purpose it is not necessary to use the default user's mail account, nor any one of the others. On the contrary this would have created the problem to hide outgoing mails of the user. We need only to create a mail message with information about the device, and send those to the attacker. Then we include in Hypnotoad the open source JavaMail API library. With this we can able to send mail messages through the infected mobile device. It is important to confirm that we do not use the user's mail. Thanks to *javax.mail* package we can set a default mail account that sends mails to itself or other preset mail boxes.

To show how simple can be retrieve the list of installed applications, we report the code we used.

Listing 6.1: Getting Installed Apps' Code

```
//: Malware that save installed applications
String installedApp = "";
PackageManager pM = getPackageManager();
```

```
List pack = pM.getInstalledApplications(PackageManager.GET_META_DATA);

for(ApplicationInfo pK : pack){
    installedApp = installedApp+"Package Name: "
        + pK.packageName
        + "; Source Dir: "+pK.sourceDir
        +"; Process Name: "+pK.processName+"\n";
}
```

On the other side, for obtaining the list of running processes, we simply use the command *top* from the Linux Kernel.

6.1.2 Caller Details

This application shows contact details such as email, address, organization, note. . . in a *toast*, pop-up, message whenever there is an incoming call. If there are information not set for these contact, toast will not be shown. We inject a malicious code in this app that exploit the *Read Contacts* and *Get Accounts* permissions to collect data about device contacts. We add the *Network Communication* permission and we write a code to generate phishing mail messages. We use the same technique saw in the Hypnotoad description to send mail to all the contacts available on the device. We use an external mail address and send an invitation to visit a web page. We also add information about the proprietary of the infected device to try to persuade the victim to click on that link. Every time the user receive a call, our malware reads all the information about his/hers contacts. Then identify which one have an e-mail contact, and create a message with default text to send to that address. We do not really send mail messages to all contacts present in the infected device, because we do not want to send phishing mail to our friends. We simulate this behavior sending 3 mail to an ad hoc created email address, and adds the e-mail contacts we read to the body of the message. This way we can obtain a similar comportment of the infected application without spam our contacts. The code for retrieve mail information about contacts is simple as above and it is composed by few rows. This is the core of this code:

Listing 6.2: Getting Mail Address

```
//: Malware which reads saved mail contacts on the phone
email = emailCur.getString(
    emailCur.getColumnIndex(
        ContactsContract.CommonDataKinds.Email.DATA
    )
);
```

Where *emailCur* contains the result of a query.

The application, to implement it, needs to have access to the *Read Contacts* permission, otherwise the application crash as it started.

We wrote also a variation of this malware which sends text messages instead of phishing mails. In this case we surely have more contacts to target and more chance to infect other devices. In this case we have to add the *Send SMS* permission to the

Manifest file.

6.1.3 Battery Widget

As the name suggest, this application is a widget that monitor the state of the battery of the device. We infect this with a code which try to fill RAM memory. As we mention each Android application is inserted in a sandbox. Each sandbox comes with fixed privileges and fixed RAM memory space assigned to it. Until the Android version 4.3.* this vale was fixed to max 24MB. Instead, From Android KitKat (version 4.4.4 and above) this limit is extended to 42MB.

Our malware try to fill all the memory assigned to its sandbox by the Dalvik virtual machine. We write a simple code that adds elements to a Java object list. We insert it in an infinite loop which is called every time the widget gets an update from the battery. In this malware we want to simulate the behavior of an application that needs to works with big variables, so we do not have to add any permission to the original manifest. We have to avoid the security control of Android about infinite loops, but finally the application works fine. We create an application which fill all the memory available for its sandbox through the creation of twelve threads which create and enlarge a variable. When the memory is full, the malware stops and it restarts on notification of battery status change. We write this malware because we are interested to know if the amount of used RAM, and the way in which it is used by an application, may be an indicator of the presence of some kind of malware. Unfortunately, as we show in the next chapter, this value results difficult to interpret. However we have been able to make some assumptions.

6.1.4 Ministock

This malware is similar to the one we use to infect Battery Widget, but in this case, we focus on the storage memory. We wrote a malware which simulate the behavior of a malicious code which needs to storage big quantity of data in the infected device.

Ministock application reads economics' data from the web and shows to the user if his/hers preferred stock options are increasing or decreasing. For doing that, the application needs to connect to an internet server and every time it do this we run our malicious code. We start a new thread while the application updates its data, and we save in the default storage memory a big text file. We believe that if we monitoring the free space of the devices we may be able to identify some suspicious behavior.

6.1.5 24h Analog Widget

24h Analog Widget is a really simple application of a clock. We injected in it a great number of malicious codes. We wrote code for real malware situation like the Ministock example. A malicious application that turn on the microphone and registers ambient sounds or conversations. Then it stores those in the storage memory and every interval of time is it possible to sends those through the net to the attacker. He makes two distinct version of this malware, one which registers sound continuously and one which do it every 120 seconds.

We thought also about the possibility to identify malware which intensively uses the CPU, like a miner. This kind of malware uses the device's CPU to compute Bitcoin. Monitoring the CPU values we thought we are able to identify this malicious behavior. To simulate the miner, the widget has been infected in such a way that it elaborates the decimal digits of π for each minute's update of the application.

He also writes a code which simulate calls and text message sending in a way completely transparent for the user. He wrote two distinct malware in order to ease our analysis. Furthermore, calls and text message sending can be maintained in background of Android application. In this way an attacker can do phone calls to premium number without user notice. In the same way, the attacker can send text messages to premium number without any notification by the Android operating system. This is the code that sends sort text messages

Listing 6.3: Code Sanding Messages

```

//: Malware which sends SMS to a prefixed number
SMSManager sm = SMSManager.getDefault();
sm.sendTextMessage(number, null, message, null, null);
...

```

Cause this action is system inhibited, we have to add to our application the *SMS Send* permission. Also, if we want to have the possibility to intercept provider alert about the costs of this action, we need to add the *SMS Read* permission. With this permission we are able to hide, and delete incoming messages from victim's phone provider.

This description can be applied to the calls malware. The code in this case is different, but quite simple too.

Listing 6.4: Calling Code Sample

```

//: Malware which makes a call to a prefixed number
Intent callIntent = new Intent(Intent.ACTION_CALL);
    callIntent.setData(Uri.parse("tel:123456789"));
    startActivity(callIntent);

```

Obviously, we need to add a permission here. We need the *Call Privileged* or also the *Call Phone* privilege to make a phone call without user confirmation.

We further stress that with root permission, all the Android permissions need to perform the above tasks could be bypassed.

6.2 Analysis Software Introduction

In this section we present the data we obtain from our analysis. We collected data from various applications (normal and infected) and the we analysed the direct comparison in order to show some common behavior of malware.

Combining the mobile-side and the server-side of ARAM we are able to create an unique "sign" of each application installed on the device. In fact, if we install ARAM on a clear device we can figure out the normal behavior of it. After that, if

we install a new application we are able to identify its behavior comparing the old and the new snapshots of that device.

6.2.1 SimpleAnalyzer Tool

The static analysis software, SimpleAnalyzer, is a Java program which permit to analyze data stored in the ARAM's server. It permit to load values from two different period of time and compare their evolution in time. These values can belong to the same user or to two different users.

The analysis software provides two visual representations: numeric data presentation and graphical. We focused our attention on the mean values, standard deviations, area's area and values oscillation (further described later). Overall we analyzed the processor, the system memory, the storage memory, the battery power, the number of text messages and calls values. We also consider mean values and standard deviation, globally, when the screen was ON and when it was OFF. With this distinction we are able to identify, for example, if there is an active service which use intensively the processor, on the device.

Also, the analysis is made on the amount of data traffic generated when the screen is ON and when it is OFF. This way, we are able to identify suspicious actions of some applications. In fact, if an application sends a great amount of data when the screen is OFF, it is probably that this application is doing something malicious. For example, if a social application like Facebook sends great amount of data when the screen is OFF, it is probably that the user is unaware of what this application is doing, then we can consider this as a suspicious action. However, there are some distinctions to do. A cloud storage application, such as Dropbox, synchronizes many data between the cloud and the device. This action requires some times and often the screen went OFF during the synchronization process. Because of this, we consider also the type of the default connection available. In fact, many applications which requires a great amount of data transfer make this only when a Wi-Fi connection is available.

The analysis software use the screen ON/OFF techniques also with the calls and the text messages. In this case is simpler to identify a malicious behavior, because it is not common for users to make phone calls and send SMS messages without the use of a screen. Then, this actions are considered highly suspicious. Comparing the list of active application on the device with other states of the same device and with the data of other users, analysis software is also able to identify which application makes the malicious action.

SimpleAnalyzer does some normalization work while comparing different logs. For example, if we analyze stock of states collected every 5 seconds with other collected on intervals of 30 seconds it is obvious that we obtain incomparable data. The difference in the data sent between two states of the first stock is probably less than the one present in the second block of data. However, this does not mean that the second data are suspicious. For this reason, we use normalization techniques which makes all data comparable. Often, normalization consists in percentage values or in values weighted on number of data analyzed, such as for the Oscillation Coefficient.

6.2.2 Oscillation Coefficient

Observing the graphs we have obtained during a preliminary analysis, we noticed a greater values fluctuations in the infected application. We consider to use a function which can evaluate the length of the graph, normalized on the number of elements.

$$\frac{1}{n} \cdot \sum_{i=1}^{n-1} \text{mod}(x_i - x_{i+1}) \quad (6.1)$$

Where n is the number of rows in the database and x is the value of the data.

This value approaches to 0 if there are no difference in consequent values, so if there are no fluctuations in the graph. High values in the coefficient means high oscillation factor. The oscillation coefficient combined with the mean values give us the tool for understanding how the application uses CPU and RAM. We can read this data as follows:

- Same average values and oscillation coefficient indicate similar behaviour in the two applications, respect a specific resource;
- Same average but different oscillation coefficient indicate that one of the two applications has a discontinuous usage of the same resource;
- Different average and different oscillation coefficient indicate different behaviour in the two applications, respect a specific resource.

6.3 Test Samples

The logs file (collection of snapshots) we used for this analysis were obtained through ARAM mobile application. We install ARAM's client on a device, then we installed an open-source application and registered the device's behaviour. We run ARAM for approximately one week. After that, we removed the open-source application and we installed the same one, but this time infected with a malicious code. Again, we collected data through ARAM for approximately one week. We used many malwares and open-source applications for our studies, in this work we will present the most relevant. We will analyse the data collected and for each study we will focus on what the malicious code introduced in the device's behaviour.

6.3.1 Battery Widget Data

Now we present the data obtained by the comparison of the normal application, Battery Widget, and the infected one. We infect this application with a code which fills the sandbox's RAM memory assigned to the application.

In table 6.2] we show the behavior of the application in the infected and non-infected states obtained with the analysis software we develop. We start with a summary of principal resources, CPU and RAM usage. We have to the left side the CPU usage and to the right the RAM usage. Each figure is divided in two columns. In the first one we find the value of the non-infected database, and in the right one, the infected values.

CPU			RAM		
<i>Mean Values</i>					
	Normal	Infected		Normal	Infected
Global Mean	35.43	75.21	Global Mean	82.16	81.74
Screen On	62.86	86.49	Screen On	81.63	81.42
Screen Off	32.73	72.34	Screen Off	82.22	81.82
<i>Standard Deviation</i>					
	Normal	Infected		Normal	Infected
Global	15.61	28.64	Global	1.3	1.85
Screen On	34.51	22.56	Screen On	1.28	2.3
Screen Off	12.26	29.99	Screen Off	1.31	1.72
<i>Oscillation Coefficient</i>					
	Normal	Infected		Normal	Infected
Global	12.0	12.88	Global	0.15	0.86
Screen On	1.96	2.25	Screen On	0.04	0.29
Screen Off	10.04	10.63	Screen Off	0.11	0.58
<i>Graph Area</i>					
	Normal	Infected		Normal	Infected
Global	35.39	75.17	Global	82.11	81.7
Screen On	59.8	86.68	Screen On	81.59	81.42
Screen Off	33.03	72.29	Screen Off	82.22	81.82

Table 6.2: Battery Widget global CPU usage graph

Looking to the data, we can affirm that the CPU usage in the infected case is higher than the normal usage. In fact, the global CPU usage switches from 35.43 to 75.21 percentage. But the more significant value is represented by the gap between the CPU values when the screen is OFF. We have approximately 40 percentage points of difference. This suggest that the application works heavily even when the user is not using the device.

The standard deviation values are generally higher in the infected application, and are also generally high. This means that the device have a great oscillation in the usage of the CPU. In fact, we can see that the oscillation coefficient is 12 in the normal application and 12.88 in the infected one. In the next image we can see difference between the two CPU global usage graphs. Observing the global oscillation coefficient, we are able to say that both the two applications have a great number of peaks and valley. But combining this value with the global mean values, we can say that in the first case the fluctuation vary among small values instead of the infected case. This results is confirmed also by the difference in the area of the two graphs. Looking at the Fig.6.11 we can instantly realize the amount of the difference in the usage of this resource. A constant behavior like the one showed by the infected

application on the CPU usage suggests the presence of an application which is doing something suspicious. The RAM values, instead, are more similar between the two applications. The value which has the higher difference is the oscillation coefficient. In Fig.6.12 we compared the graphs of the RAM usages. The infected application has more fluctuations in the percentage values.

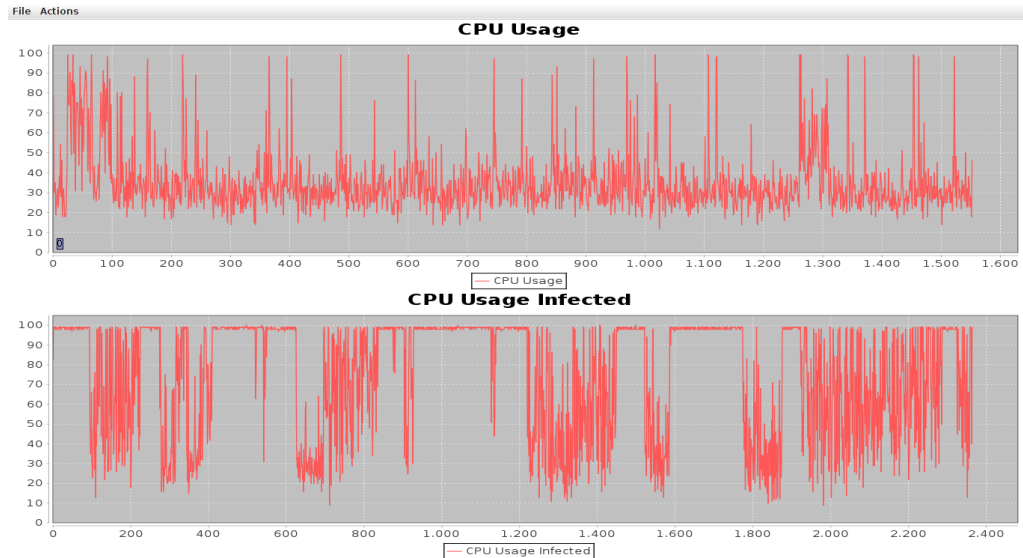


Figure 6.11: Battery Widget global CPU usage graph

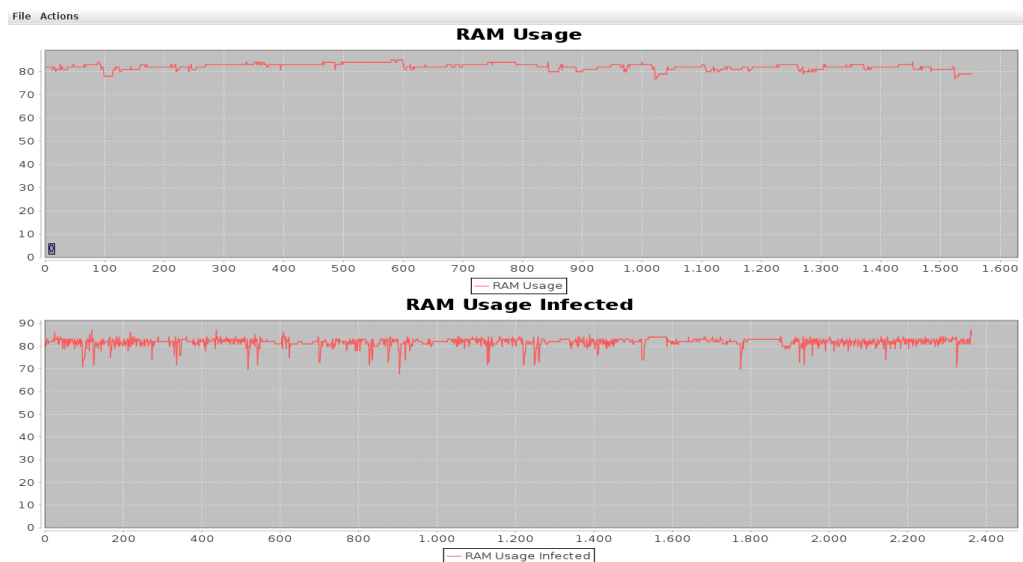


Figure 6.12: Battery Widget global RAM usage graph

This is compliant with the infection. In fact, Battery Widget was infected with a code which fills the application sandbox memory. When it is full, the Android system tries to free up memory space with its own methods. This causes the application's pause and the emptying of part of the application memory. So we can use this value as

marker for situation like this. However, we expect also a not so great difference in RAM values since Android shut in a sandbox all the applications and control the utilization in time. Then, even if the malware works correctly it can fill only the memory assigned to its sandbox, which in this test was fixed by Android around to 26 megabytes (a common values among all the applications). The last important value for our analysis of this malware, is the battery usage. In the following graph, Fig.6.13 we can see how much battery the infected application consumes. Until the thousandth state, the phone with the non-infected widget was under charge. Instead, the infected one was battery powered from the beginning. However, after the charge is completed in the normal application, we can compare the slope of the graphs. From then, we can see that we have an higher battery consumption in the infected case. This is accordant with the other data we present. In fact, a greater use of the CPU surely causes a faster battery consumption.

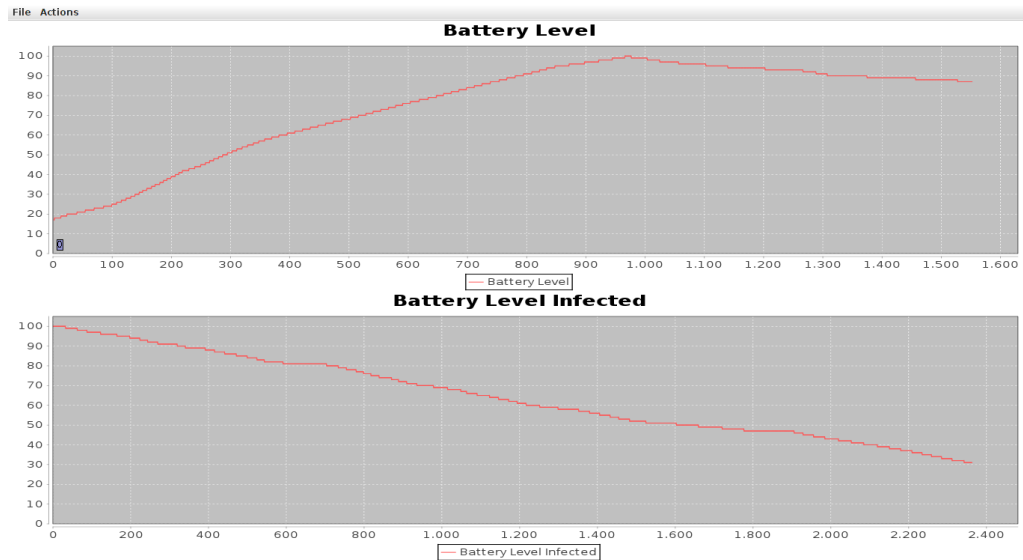


Figure 6.13: Battery Widget global battery usage graph

6.3.2 Caller Details

We compare the data collected on Caller Details application. This application was infected with a code which steals address book information, and sends those through internet to a mail address. This malware is activated only when the device receives a telephone call. We expect high values when the phone is ringing and during the call. The table 6.3 shows the CPU's and RAM's values we collected.

CPU			RAM		
<i>Mean Values</i>					
	Normal	Infected		Normal	Infected
Global Mean	33.72	55.1	Global Mean	81.48	81.7
Screen On	63.36	81.47	Screen On	82.54	81.22
Screen Off	31.77	50.9	Screen Off	81.41	81.78

<i>Standard Deviation</i>					
Global	22.09	28.66	Global	1.24	1.25
Screen On	37.96	34.15	Screen On	2.06	1.74
Screen Off	20.63	27.69	Screen Off	1.16	1.15
<i>Oscillation Coefficient</i>					
Global	15.24	24.4	Global	0.35	0.91
Screen On	1.38	3.15	Screen On	0.04	0.19
Screen Off	13.86	21.26	Screen Off	0.31	0.72
<i>Graph Area</i>					
Global	33.67	55.05	Global	81.43	81.65
Screen On	61.64	78.37	Screen On	82.51	81.23
Screen Off	31.87	51.39	Screen Off	81.41	81.78

Table 6.3: Caller Details CPU and RAM comparison

From the confront, we can observe that for the CPU's values we have a higher percentage in the infected case. We can observe also that in the battery widget case we have a much higher difference in the value when the screen is OFF. We expect that, in this case, these values are less different because this malware activates itself just when the device receives a call, then when the screen is ON. It sends the personal data through internet after that event, then the screen may be ON or OFF.

However the CPU graph in Fig.6.14 shows a behavior similar to others cases of study for the infected application.

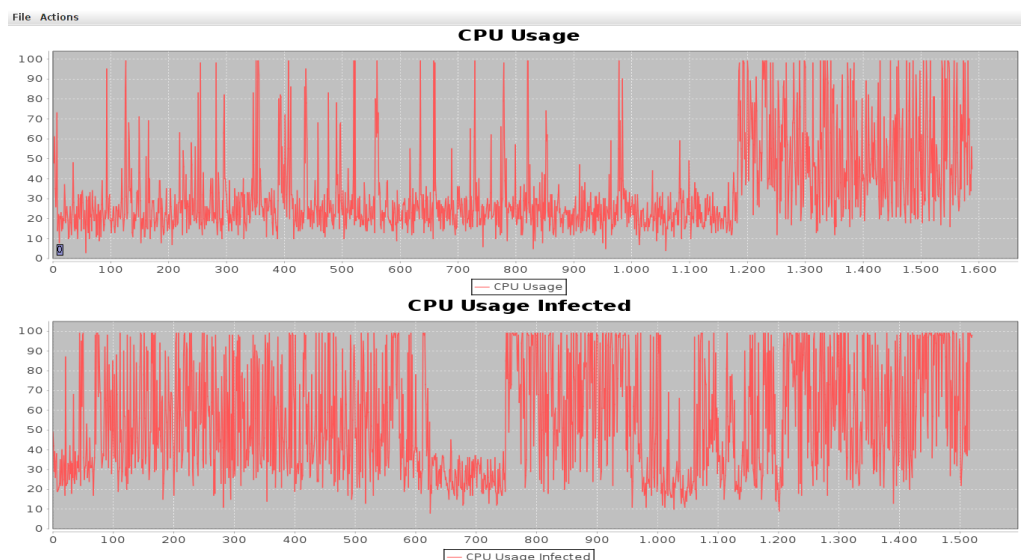


Figure 6.14: CallerDetails Global CPU usage graph

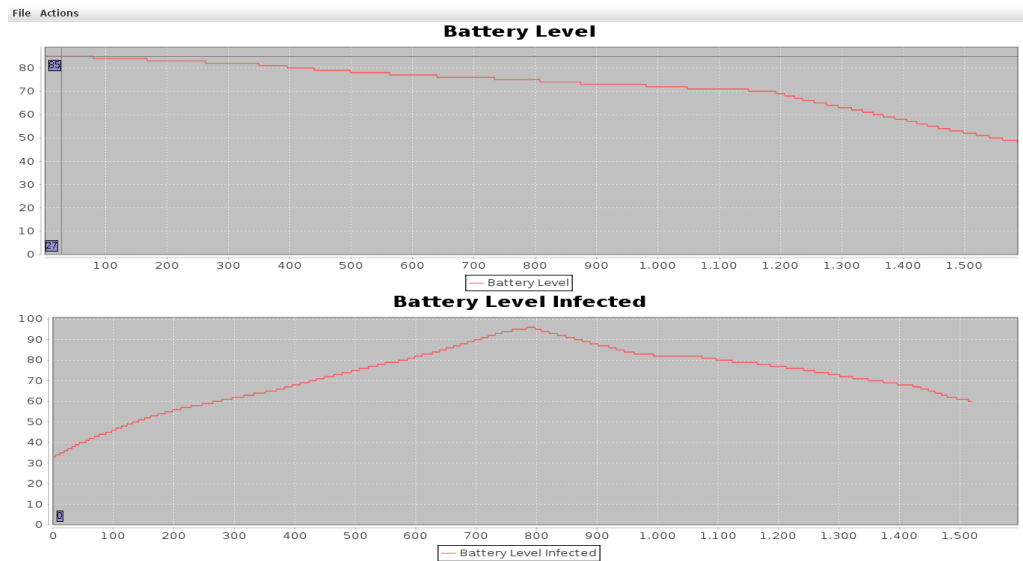


Figure 6.15: Caller Details Global Battery usage graph

From the table 6.3, we can see that infected application has higher fluctuations identified by the oscillation coefficient. Also, combining this value with the global CPU usage, we are able to say that infected application has higher values. This fact is confirmed by the area in the graph, which is higher in the infected case.

As in other study cases, the values of the RAM are not so different in the normal and infected application. The only significant difference is given by the oscillation coefficient which is 0.35 in the non-infected application and becomes 0.91 in the infected one. However the mean values, and also the standard deviation ones, are approximately the same. This suggests us that Android manages the memory resources always in the same way, regardless of the running applications.

The battery usage is, in this case, a valid tool to identify the presence of running malware. Obviously, alone is not enough but it can easily identify suspicious behavior. In Fig.6.15 we can observe the battery power graph slope and compare the two cases. In the infected case, after the battery charged around the eight hundredth snapshot, we can see that the consumption of energy is higher than in the non-infected application.

We also compared the amount of data transmitted and received from the application in both the cases. In the infected one we can observe higher values in both the cases. This is what we expect, because the uninfected application do not have the *Network Communication* permission. Then, the malicious application obviously generates more network traffic. This can be shown in table 6.4 while the comparison of the number of calls made, received and lost is showed in table 6.5.

Data Traffic		
<i>Hourly Mean Values</i>		
	Normal	Infected
Global Data Sent	244 kb	1 940 kb
Global Data Got	1 167 kb	504 kb

Application Data Sent	0 kb	896 kb
Application Data Got	0 kb	0 kb

Table 6.4: Caller Details Data Traffic

Calls		
<i>Both on 6 hour interval</i>		
	Normal	Infected
Made	2	1
Received	4	6
Lost	0	5

Table 6.5: CallerDetails number of calls

6.3.3 Hypnotoad

This application is a live wallpaper and was infected with a code which sends personal information such as installed and running applications to an email address. In this comparison we obtained different results than that in the other cases. First of all the CPU values are not in line with the others. As we can see in table 6.6, generally the values are higher in the uninfected case instead of in the other one. Also, the oscillation coefficient is lesser in the infected application, even if only slightly. The values vary from the 26.99 of the normal application to the 24.04 of the infected one. We can graphically see this result in Fig.6.16

CPU		
<i>Mean Values</i>		
	Normal	Infected
Global Mean	72.67	70.85
Screen On	84.53	84.4
Screen Off	71.49	68.19
<i>Standard Deviation</i>		
	Normal	Infected
Global	26.06	27.71
Screen On	23.01	22.83
Screen Off	26.34	28.57
<i>Oscillation Coefficient</i>		
	Normal	Infected
Global	26.99	24.04
Screen On	1.65	2.7

Screen Off	25.34	21.33
<i>Graph Area</i>		
	Normal	Infected
Global	72.62	70.81
Screen On	84.03	83.47
Screen Off	71.53	68.37

Table 6.6: Hypnotoad CPU values

This fact, combined with the global CPU usage values, result in a more similar power battery consumption among the two applications. We can see in Fig.6.17.

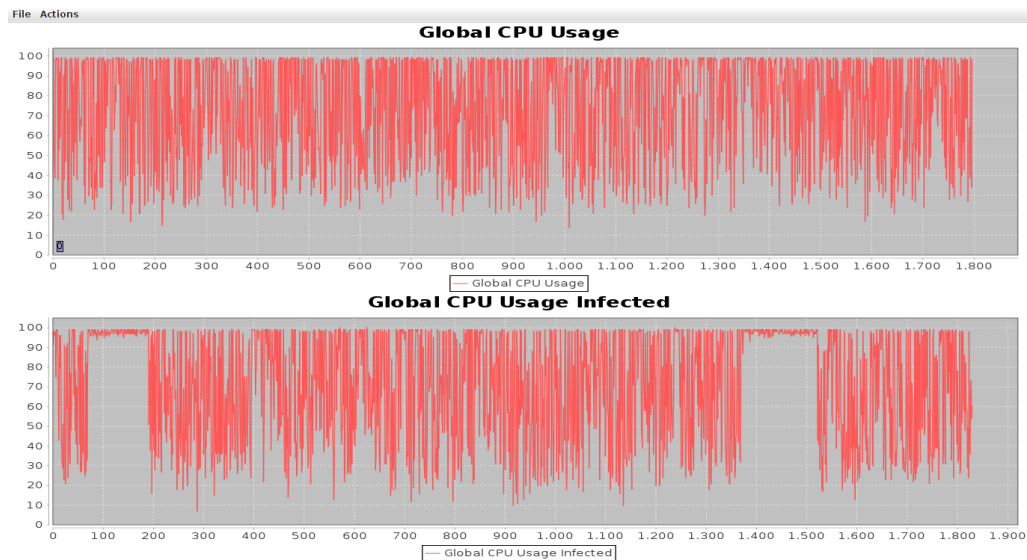


Figure 6.16: Hypnotoad CPU graph

The slope of the graphs are almost the same for both applications. We can see similar slope, either when the device is on charging either when it is discharging. Probably, the behavior of the device, when it is not infected, is caused by other non malicious service running in background. We identify a constant network traffic by another application installed on the phone. In fact, during the recording of the non infected device, we identified that process *com.zeptolab.ctr.ads* had constantly transmitted data. This means that this application runs a service which influenced our analysis, and it caused a behavior similar to the infected one. This also could indicate that the application executes some non requested operation and for this reason it might be considered malicious.

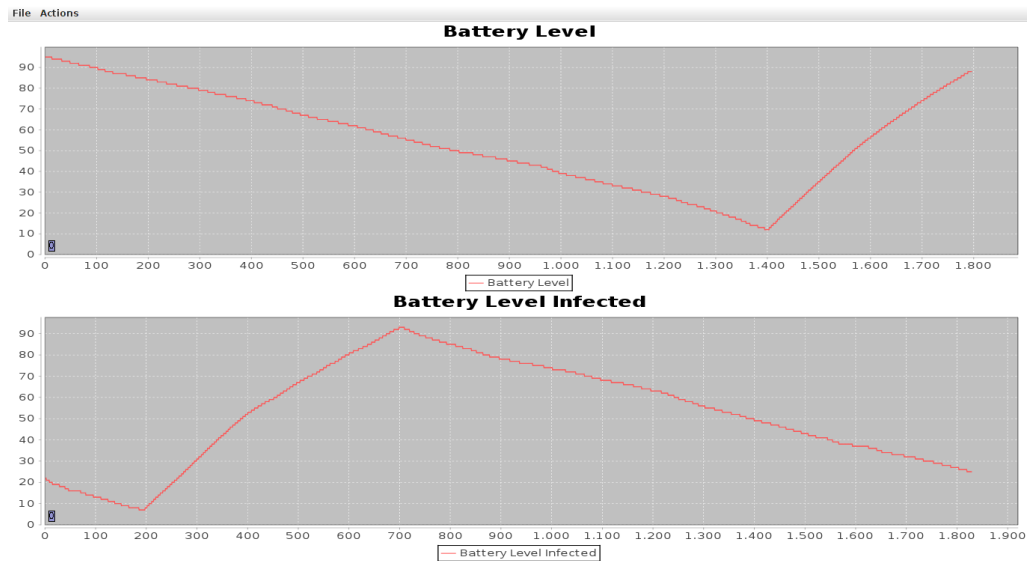


Figure 6.17: Hypnotoad global battery usage graph

6.3.4 24h Analog Clock

24 hours Analog Clock is a clock widget, infected by numerous kind of malware. It is important to say that the data which we will show in this test is obtained using a different device from the precedent tests.

We compared the normal application against one infected with a miner. In this case we identify a similar behavior with those shown in Battery Widget and Caller Details sections. In table 6.7 we can see that values in the infected application are higher than the uninfected version. The oscillation coefficient and the global mean value of the malicious application show high frequency changes.

CPU		
<i>Mean Values</i>		
	Normal	Infected
Global Mean	21.02	46.93
Screen On	33.6	48.97
Screen Off	18.69	45.96
<i>Standard Deviation</i>		
	Normal	Infected
Global	9.38	27.13
Screen On	19.24	25.04
Screen Off	5.98	28.07
<i>Oscillation Coefficient</i>		
	Normal	Infected
Global	6.43	26.67
Screen On	1.75	8.64

Screen Off	4.68	18.03
<i>Graph Area</i>		
	Normal	Infected
Global	21.0	46.88
Screen On	31.15	47.81
Screen Off	19.14	46.51

Table 6.7: 24h Analog Clock miner CPU values

In Fig.6.18 we can see this result. We can also observe that the battery power consumption is comparable to other study cases. In Fig.6.19 we can see that the graph slope is greater in the infected application. It is important to stress that these results are obtained running ARAM on a different device, but they are equally comparable.

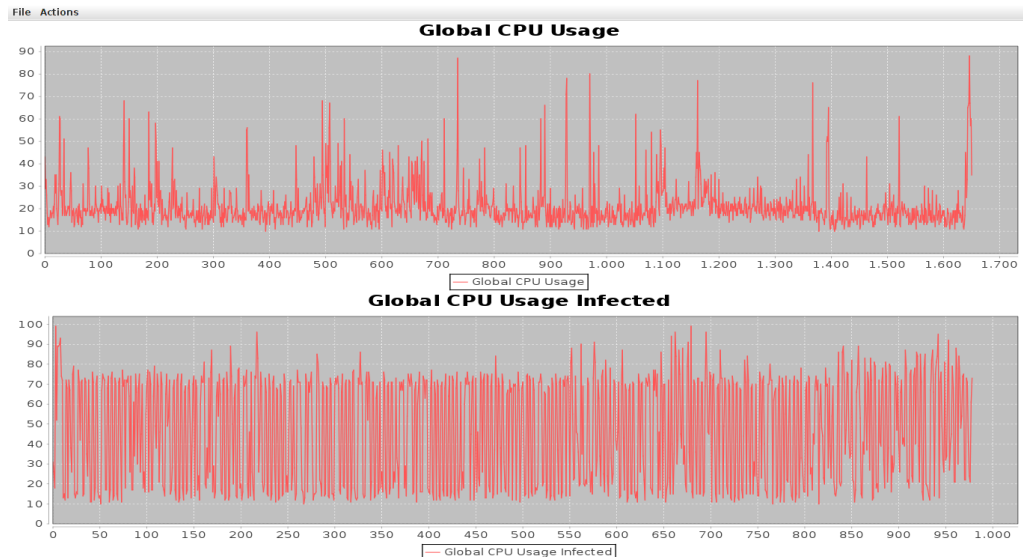


Figure 6.18: 24h Analog Clock miner global CPU usage graph

The next data comes from the comparison of the uninfected application and the same but infected with a service that register environment sound every 120 seconds. We can observe that the CPU usage values are comply with the other data we collected. In table 6.8 we can see the CPU behavior through six hours of monitoring.

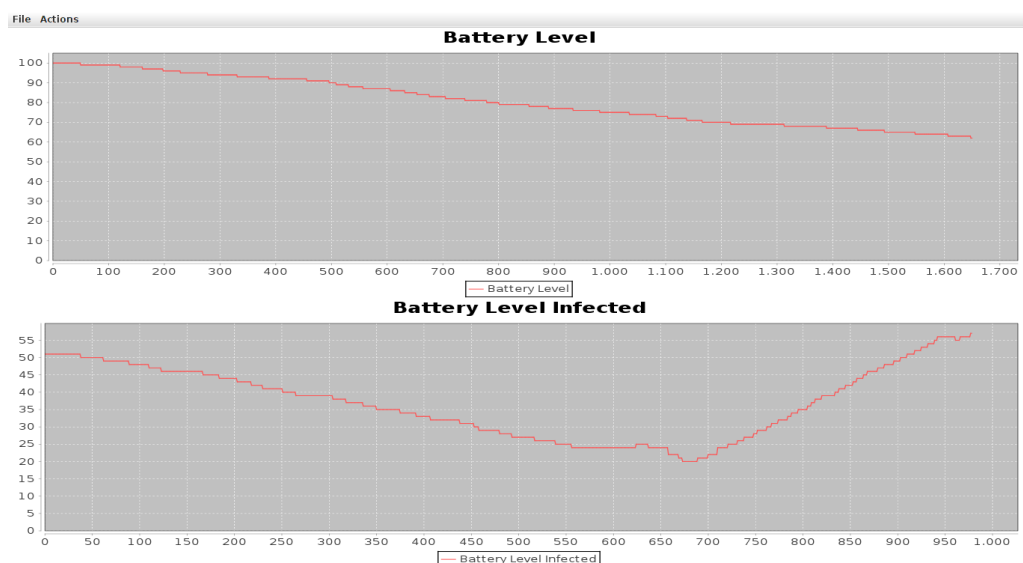


Figure 6.19: 24h Analog Clock miner global battery usage graph

CPU		
<i>Mean Values</i>		
	Normal	Infected
Global Mean	21.02	25.06
Screen On	33.6	39.64
Screen Off	18.69	23.77
<i>Standard Deviation</i>		
	Normal	Infected
Global	9.38	12.25
Screen On	19.24	23.72
Screen Off	5.98	14.26
<i>Oscillation Coefficient</i>		
	Normal	Infected
Global	6.43	12.77
Screen On	1.75	1.23
Screen Off	4.68	11.55
<i>Graph Area</i>		
	Normal	Infected
Global	21.0	25.03
Screen On	31.15	38.25
Screen Off	19.14	23.89

Table 6.8: 24h Analog Clock sound recorded CPU values

The left side of this table represents the values of the application not infected and are the same as the previous example. This data suggest that infected application uses the CPU more often. In the next graph, Fig.6.20, we can see how the CPU resource is used at regular intervals. Knowing what the malware is supposed to do, we can see clearly when it performs the registration.

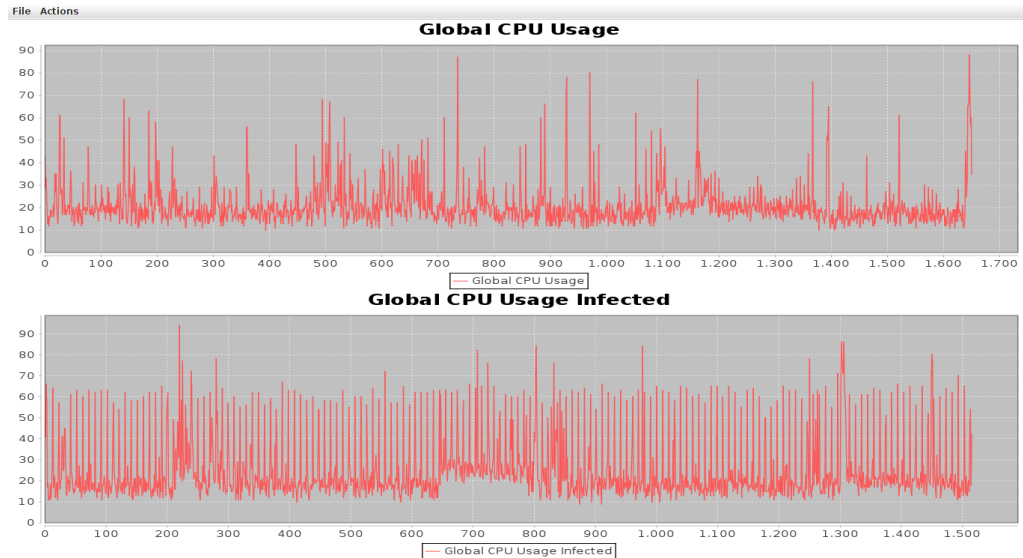


Figure 6.20: 24h Analog Clock sound recorded global battery usage graph

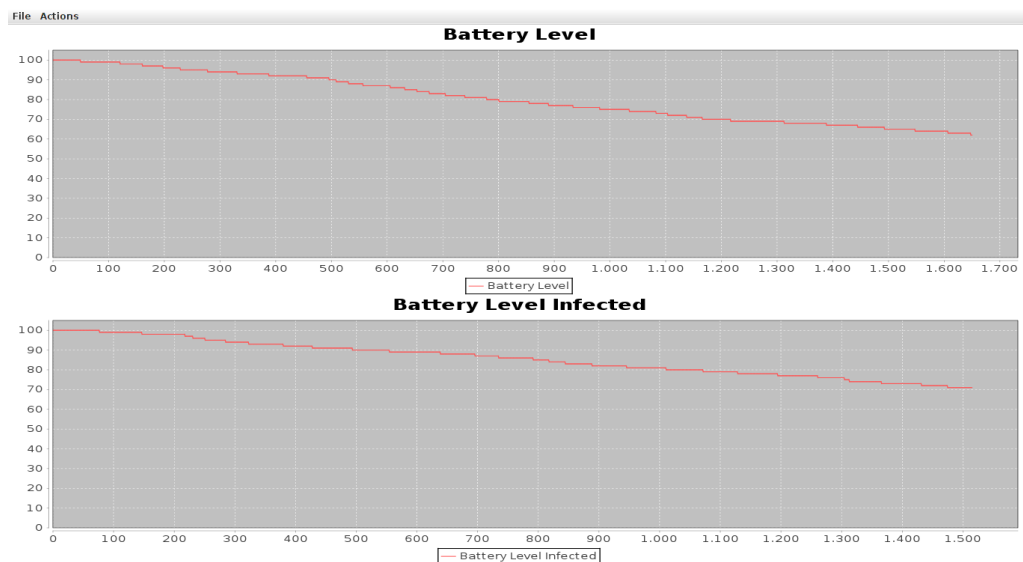


Figure 6.21: 24h Analog Clock sound recorded battery usage graph

However, in this case the power battery consumption results less with the infected application. This can be caused by the difference in amount and frequency of data got by the device in the two cases. In fact, the non infected application receives from mobile network data requested by another installed application, more often than

in the infected application. This causes a greater consumption in battery power for the uninfected application. The synchronization of this application with a server causes a greater usage of the phone antenna and a greater power consumption. In the previous case this difference was meaningless because the CPU usage in the infected case was significantly higher. Fig. 6.21 shows the power battery usage. As we can see, the difference for the two cases is about six percentage points. This fact is discordant with the other example we presented, where the battery power consumption was greater. However, as we said, this difference may be caused by the network data traffic.

The next data we present are obtained by the comparison of the normal application 24 hours Analog Clock with one infected with a constant sound recording. Unlike the previous case, this data are more similar to the other malware data we have shown. Now we briefly present these data in table 6.9. We may expect that these data are similar to the previous one, however they are more different than we can though. Even if the malicious behavior is almost the same, this malware use more intensively the principal resources. As we can see in Fig.6.22 and 6.23

CPU		
<i>Mean Values</i>		
	Normal	Infected
Global Mean	21.02	59.63
Screen On	33.6	63.8
Screen Off	18.69	59.3
<i>Standard Deviation</i>		
	Normal	Infected
Global	9.38	12.37
Screen On	19.24	17.79
Screen Off	5.98	11.84
<i>Oscillation Coefficient</i>		
	Normal	Infected
Global	6.43	10.41
Screen On	1.75	1.09
Screen Off	4.68	9.32
<i>Graph Area</i>		
	Normal	Infected
Global	21.0	59.6
Screen On	31.15	61.74
Screen Off	19.14	59.47

Table 6.9: 24h Analog Clock constant recorded CPU value

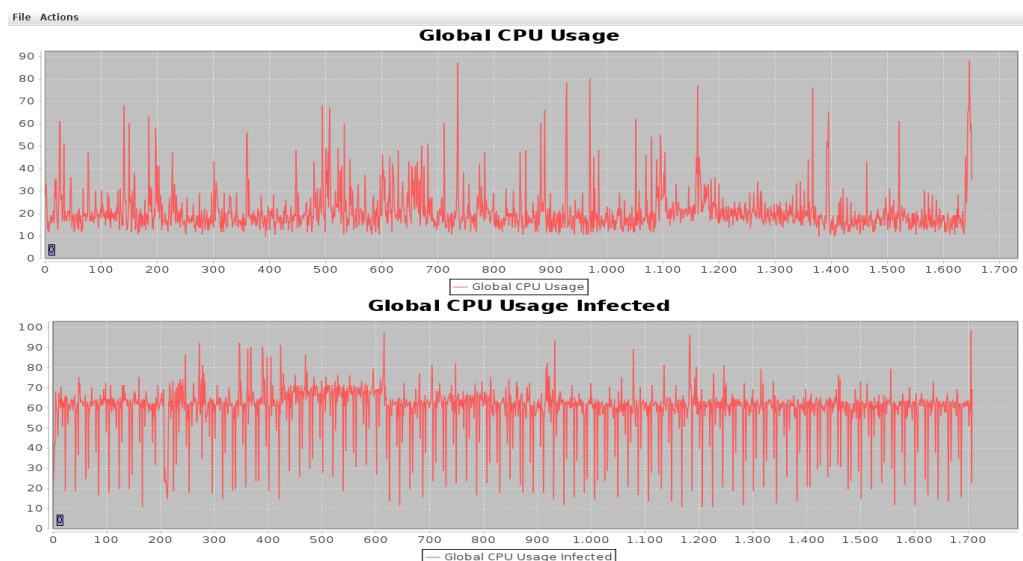


Figure 6.22: 24H Analog Clock constant sound recorded global CPU usage graph

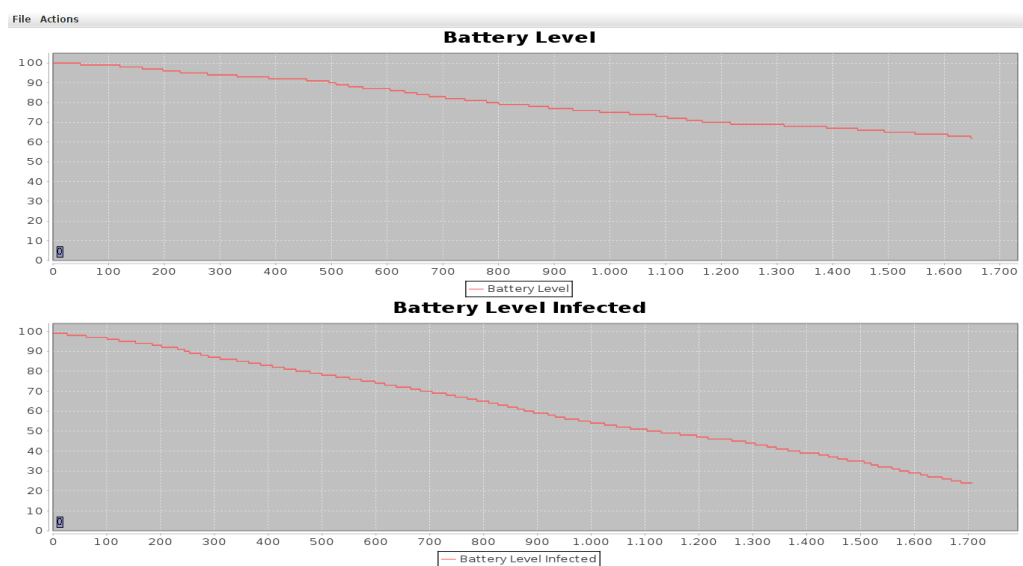


Figure 6.23: 24H Analog Clock constant sound recorded Battery usage graph

As we can expect, these values are higher in comparison to the previous. Also, the graph's slope in the battery power is once again higher in the infected application. This is caused by the greater CPU usage. The last data we analyse is the silent call malware. This malware permors uncontrolled phone calls to a preset phone number. This malware presents a behaviour like the malware which records the sound every 120 seconds. Table 6.10 shown the results.

CPU		
<i>Mean Values</i>		
	Normal	Infected
Global Mean	21.02	22.51
Screen On	33.6	39.73
Screen Off	18.69	20.86
<i>Standard Deviation</i>		
	Normal	Infected
Global	9.38	13.28
Screen On	19.24	24.15
Screen Off	5.98	11.73
<i>Oscillation Coefficient</i>		
	Normal	Infected
Global	6.43	10.24
Screen On	1.75	1.27
Screen Off	4.68	8.96
<i>Graph Area</i>		
	Normal	Infected
Global	21.0	22.48
Screen On	31.15	37.36
Screen Off	19.14	21.09

Table 6.10: 24h Analog Clock silent calls CPU values

The graph for this test study is shown in Fig.6.24 and looks like the one in Fig.6.20. This is interesting because this malware does a different malicious action but it acts on CPU usage in similar ways.

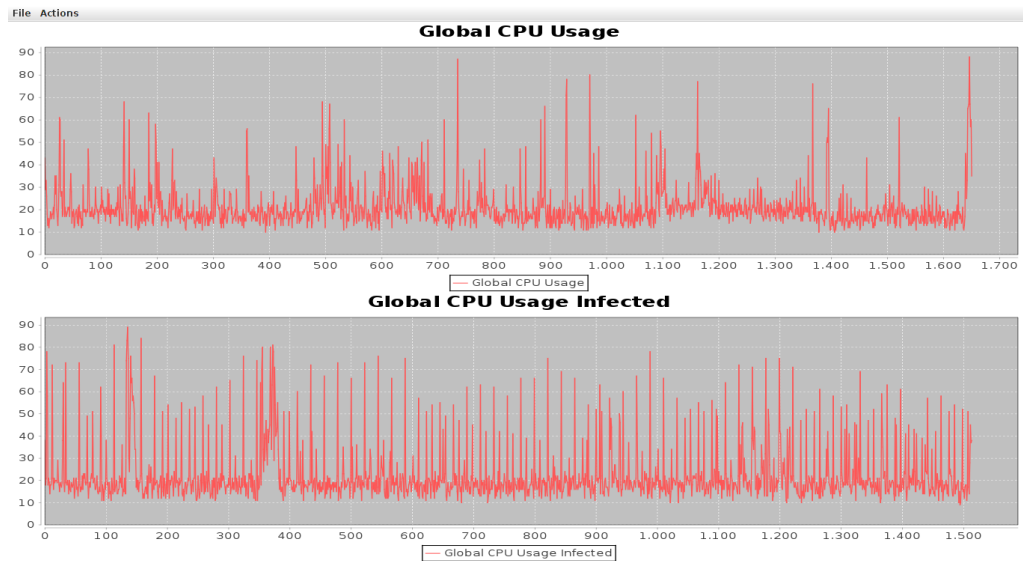


Figure 6.24: 24h Analog Clock silent calls sound recorded global CPU usage graph

The Fig.6.25, shows the battery usage confront about the two applications. In this case the graph is also similar to Fig.6.21, but in this case the battery power consumption is little higher in the infected application. This may be caused by the greater consumption of energy caused by the antenna of the device. However, also in this case, the two malware behaves in similar ways on hardware components of the device.

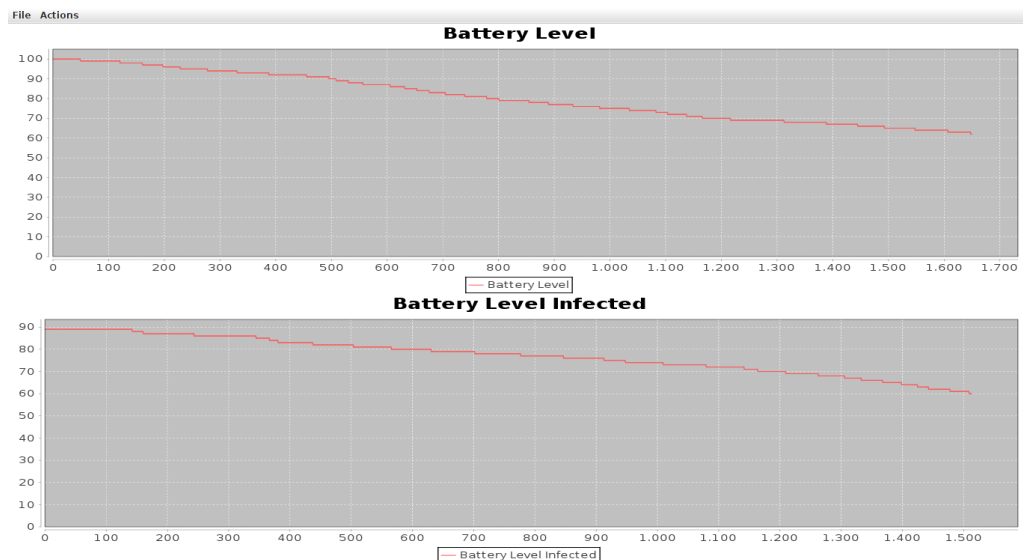


Figure 6.25: 24h Analog Clock silent calls recorded battery usage graph

For this example is also extremely important to show the difference in the numbers of calls made by the phone. In fact, for this kind of malware it may be sufficient this value to identify the infection. In table 6.11 we present these values.

Calls		
<i>Both on 6 hour interval</i>		
	Normal	Infected
Made	8	87
Received	2	0
Lost	1	0

Table 6.11: 24h Analog Clock number of calls

From the registered data is absolutely clear that something malicious is happening. The gap between the two phone calls' values is too high to be considered as normal. By ending our analysis of the Snapshot Module we can conclude that similar malicious behavior may have different impact on the resources utilization. For example, the stealing of personal information might be achieved in many ways and using different resources. Also, there are malicious actions which do not affect so much the parameters we monitor and, for this reason, they are not identifiable by our manual analysis. We obtained an important result about the RAM usage in the infected and non infected cases. By our analysis, in fact, we are able to say that RAM is almost unaltered by the presence of a malware. This result might be anti intuitive in general cases, but if we consider the sandbox structure of Android we realize that a single application could only access to a limited portion of available device memory. Also, we can easily say that the global processor usage might be not sufficient to identify malware which works only when the screen is ON. Equally, the battery power usage might be ineffective with malware which runs only when the device is under charge. For these reasons we have to consider more parameters in the policies for the automatic analysis of infections. However, we also shown how easily is find a great amount of malware. In many cases is in fact possible to use only a few number of parameters, even only one, to identify correctly an infection.

The Hypnotoad example is perfect for stress an important result. As we saw, the application has a different behavior from the others we presented. The example demonstrates that our analysis might fail on a single analysis. However it is extremely important to notice that comparing its behavior with other device uninfected behaviours led us to identify a suspicious behavior even when we did not discover it on the first analysis. This is the really an important result, because it demonstrate that is possible to use other users data to identify anomalous behaviours. The analysis we made on these test applications are then usable in real scenario on real malwares. Through the collection of many users data we are also able to make a sharpen analysis on the behaviours of each application. If we need to know if an application is doing something suspicious, we only have to compare its behavior with the one of each other users, or with a mean of those values. Then if the behavior of the analyzed application deviates over threshold values, we trigger an alarm to that user. We will focus on these aspects in the Policy and Application Modules.

[Policy](#) [Log](#)
[Average of Day](#) ▼

Cpu	Ram	Battery Level	Battery Temperature	Data For App	Calls	Sms	Data From	Data To
4	69.767426	73	28.05303		13:9.3	24:4	2014-11-06	2014-11-06

[Logout](#)

Welcome
 administrator
 administrator

Administrator Policies
 • [Show policies](#)

List of Users
 • [3.53.7](#)
 • [3.58](#)

Figure 6.6: Daily average page

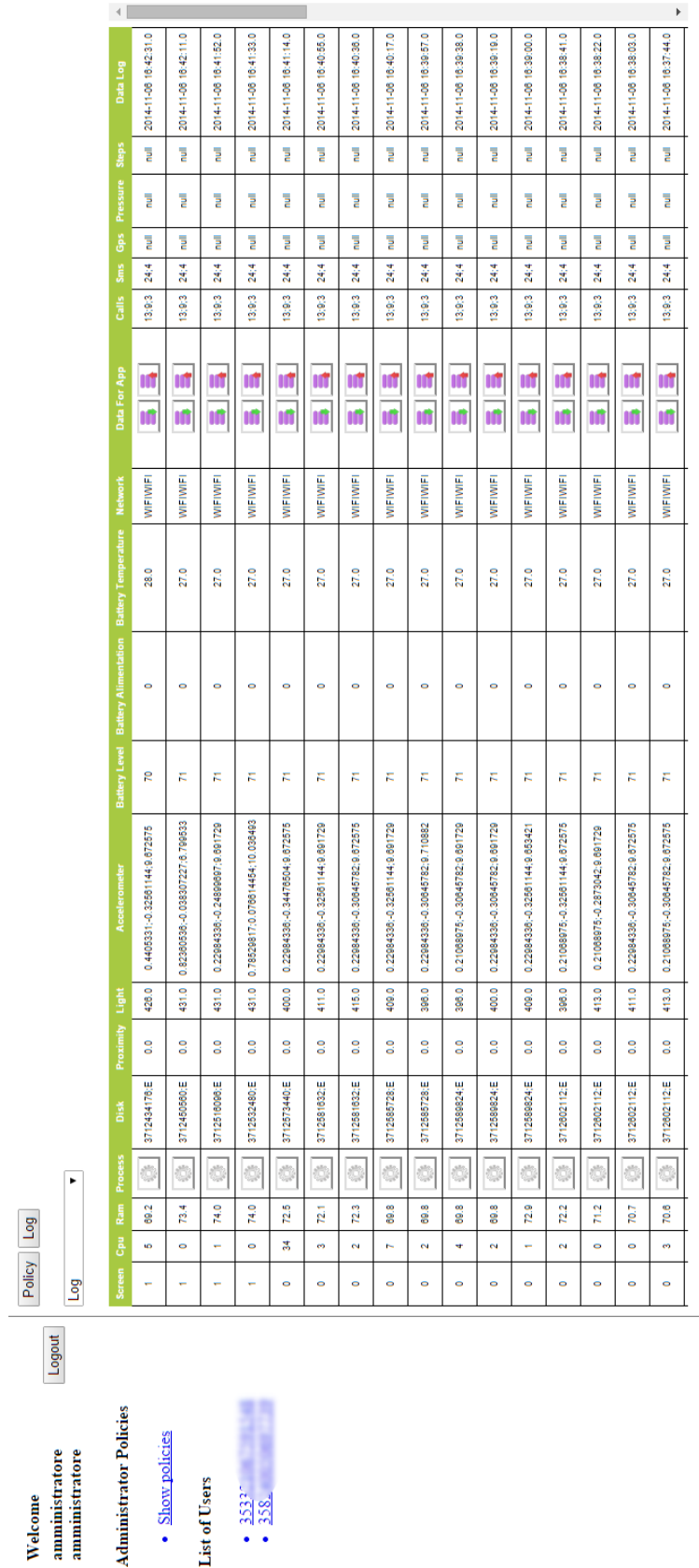


Figure 6.7: Admin server side graphic interface

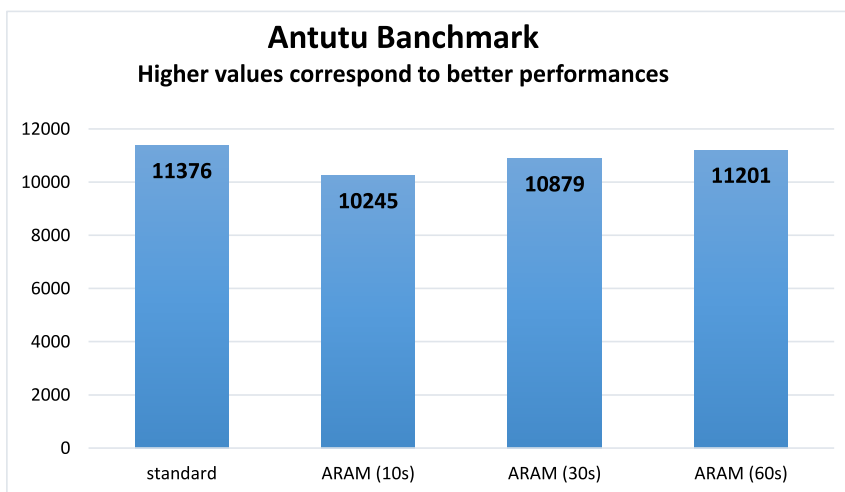


Figure 6.8: Antutu benchmark with different acquisition intervals

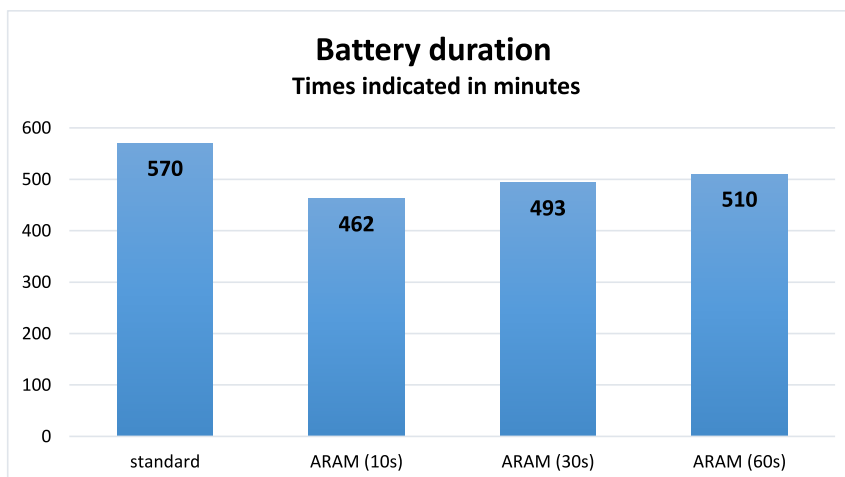


Figure 6.9: Battery duration with different acquisition intervals

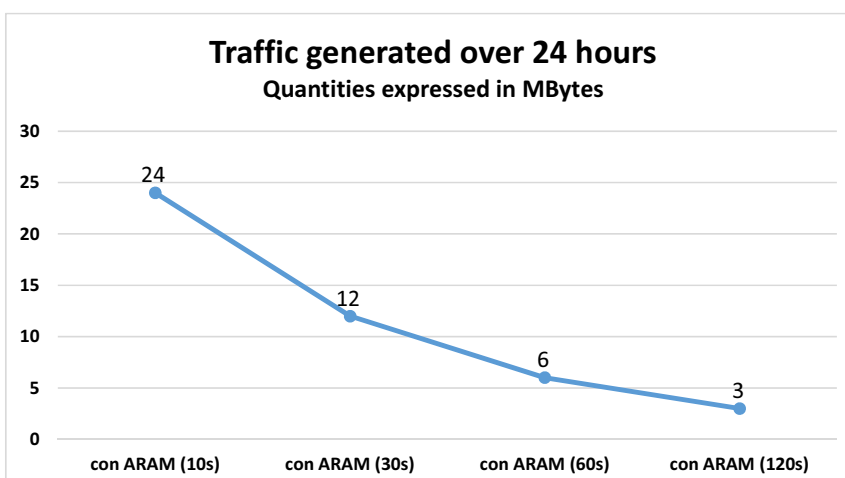


Figure 6.10: Overall traffic generated over 24 hours

Chapter 7

ARAM: Policy Module

In this chapter we will present ARAM's second module: Policy Module.

This module has a completely different compartment in respect to the Snapshot Module we saw in the previous chapter. The Policy module permits to define, as its name suggests, behavioural compartmental rules which are automatically validated by the centralized server.

7.1 The Overall Architecture

The base architecture partially overlaps the Snapshot Module scheme. This is due because Snapshot Module is the base engine which collects data and send them to the server for the storage. Then, the Policy Module works on the data already collected on the server side as shown in Fig. 7.1

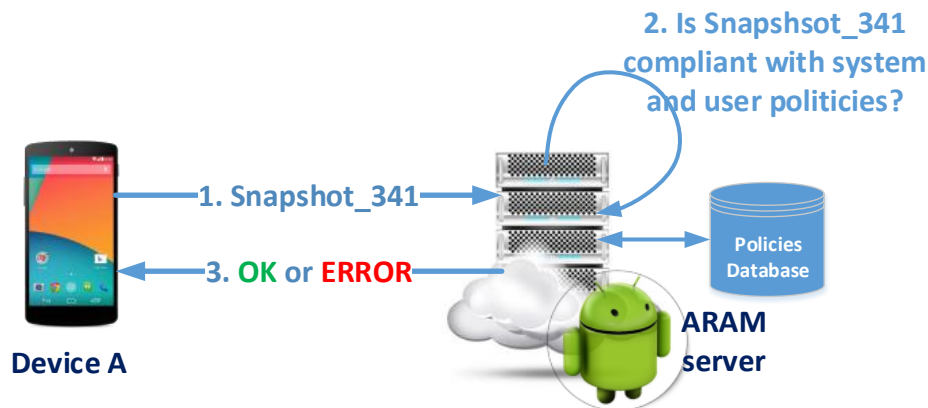


Figure 7.1: The Policy Module architecture

Every time a new snapshot is received on the server side, the module checks if the contained information and then the device behaviour is such that occurs one of the conditions contained within one of the defined policies.

One important factor to evaluate is what the system should do when a policy, wherever it is, is validated and then the device behaviour could be considered anomalous. With "policy is validated" we mean that the behaviour specified in the rule has been fulfilled. The first and easiest method could be to inform the user of its

device behaviour and let the user to choose how to act to solve the anomaly. This kind of approach is valid only if the system is able to communicate precise information about the anomaly. This could be solved by adding a precise description to any defined rule. In addition to, the user must have enough skills to understand how to handle the information received but, with an exhaustive description, the task could be really facilitated and clarified.

The second method involves to directly interact with the device's software side and to try to stop what has led to the anomaly situation. Unfortunately it is rarely possible to know exactly the anomaly cause because not all the collected information are strictly related to a specific application. For example if one rule detect that too many calls were made there is not the way to know which exact application has performed the calls. Stop the possibility to perform further calls from the device could be a too strong impediment, considering also the fact that the abnormal behaviour could be introduced deliberately by a legitimate owner demeanour.

In front of the preceding considerations, in our opinion the first approach is the most appropriate to the context. The ARAM server will limit itself to communicate the rule description to the user and the user will decide how to act. However, it could be possible to modify the system and introduce the second method as a possible choice if the first method is considered too bind to the user's skills.

7.2 Policies classes

There are two distinct sets of policies available in the system:

1. **General:** Policies defined by the system administrator. These are general rules and should guarantee a sufficient level of protection. Any mobile device owner, registered to the ARAM service, can subscribe to the whole of them or just to a subset which meets his needs;
2. **Personal:** Policies defined by the device owner. These are further rules defined directly by the user and added to the policies the server checks for him. As the name suggests, these eventual rules are valid only for the user which has inserted them.

A personal policy could be also a general policy modified by the user in some way to better represent his behaviour. Normally this happen when the rule conditions and range values, defined by the administrator, are too restrictive or even too lenient for the user.

The order in which the policies are controlled is not normally relevant and it does not affect the effectiveness of the system. But, due the fact that an user could modified a general rule, the first set to be controlled are the user's personal rules (if any is present) and then the system general rules (excluding those modified and saved in the personal set).

A second policies subdivision is based on which temporal interval the conditions target:

1. **Immediate:** the rules validate any single snapshot received. The control happens directly when the information are inserted in the database from the snapshot module. The control is instantaneous and the user could be alerted immediately if an anomaly occurs.

2. **Temporal:** the rules validate statistical values calculated over a well defined interval of time. These values are means calculated and saved from the snapshot module over time and progressively update while new snapshots are received. The control is not always instantaneous because the information arrived could be not enough for what the condition needs. The example is quite simple: if a condition targets the data received in a week and just five days are passed since the beginning then the data available is not sufficient to calculate the week average.

7.3 Policies format

Over the years, several control policies have been proposed in literature with different environments in mind. We will not use one of these because our needs are very specific and limited. What it to be controlled is just a finite set of sensors and resources available in the modern mobile devices.

Our ecosystem is so specific that the problems of putting together several policies does not exist. In fact no conflict could arise from contradictory decisions produced by different policies. In the worst case some policies could target the same sensors and could be triggered by the same values. This will result just in a series of alert message sent to the user. In any case this is symptomatic of an anomalous behaviour.

The main concern, while we were deciding how to define the policies and which format they should have in the end, was to create an easy way to translate an high level idea based on sensors values in something understandable by the Policy Module engine. Our framework is designed to reach a large share of the Android users' ecosystem and not just technicians. Then the chosen language and then the policies syntax should be easily understandable but at the same time allows to control completely the device behaviour in time.

Generalizing, a rule has this format:

```
IF "some conditions happen" THEN "act in consequence"
```

The left part is a binary expression that could be evaluate as true or false. The right part is what should be done if the left part is evaluate as true. In detail, our model, the Behaviour Policy Model, abbreviated henceforth as BePolicy, includes a simple language for expressing conditions. Our language supports only conditions expressed over two finite sets: punctual values and in-time values. The domain of the sets depends on the specific registered data. Field as the CPU and RAM, which are express by a percentage, have finite domain (integer values from 0 to 100) while information has the data sent and received over internet during two snapshots have infinite domain (from 0 to infinite). The conditions could be composed through specific logic operators to build more complex expressions. Conditions should not be confused with constraints or obligations, which are very common and relevant components in other policy languages. We don't work to prevent access to resources (constraints) or to respond with actions (obligations). We want to build an alert system and let the user acts.

DEFINITION. Let X be a set of variables over the registered values, each variable $x \in X$ has a domain Dx . An atomic condition ac definite over X has the form (x op

v) where $x \in X$, $v \in Dx$, $op \in (=, \neq, <, >)$. The conditions of the language, hence in advance BePolicy, are defined as follows:

- An atomic condition is a condition of BePolicy;
- Let c_i and c_j be condition of BePolicy, then $c_i \wedge c_j$ and $c_i \vee c_j$ are conditions of BePolicy.

The registered value which belong to X are:

- CPU
- RAM
- PROXIMITY
- LIGHT
- SCREEN
- BATTERY_LEVEL
- BATTERY_TEMPERATURE
- DATA_RECEIVED
- DATA_RECEIVED_FOR_APP
- DATA_TRANSMITTED
- DATA_TRANSMITTED_FOR_APP
- CALLS_DONE
- CALLS_RECEIVED
- CALLS_MISSED
- SMS_SENT
- SMS_RECEIVED

For all the variables $x \in X$ it is the possible to use in the conditions, instead the precise instantaneous value, the average value calculated by the Snapshot Module while the snapshots are acquired. If we consider today as day t_0 and hour 12.00 then the average are so defined:

- **avgHour0(x)**: return the hourly average for the variable x . This is calculated from hour 11.00 to hour 11.59 of current day. The value is available only 60 minutes after the beginning of the registration service in order to have a full hour data.
- **avgHour-1(x)**: return the hourly average for the variable x but for the previous hour - from 10.00 to 10.59. The value is available only 120 minutes after the beginning of the registration service.

- **avgDay0(x)**: return the daily average for the variable x . This is calculated from hour 00.00 to hour 23.59 of the day $t_i - 1$. The value is available then only 24 hours from the beginning of the registration service.
- **avgDay-1(x)**: as **avgDay(x)** but calculated for the day $t_i - 2$. This value is available then only 48 hours from the beginning of the registration service.
- **avgWeek0(x)**: return the week average for the variable x . This mean is calculated on a seven day average since the Monday of the previous week. This value is available only when a complete week of records have elapsed.
- **avgWeek-1(x)**: as **avgWeek(x)** but calculated over the week before. This value is then available only when two complete weeks, from Monday to Sunday, are passed.
- **avgMonth0(x)**: return the month average value for the variable x . This mean is calculated on the month length since day one to the last of the month. This value is available only when a complete month of records have elapsed.
- **avgMonth-1(x)**: as **Month(x)** but calculated over the month before. This value is then available only when two complete month have elapsed.

Over those average is possible to calculate the overall average in device lifetime:

- **avgHour(x)**: return the hourly average for the variable x calculated over all the **avgHour0(x)** registered in the time. It is the average over the hourly average;
- **avgDay(x)**: return the daily average for the variable x calculated over all the **avgDay0(x)** registered in the time. It is the average over the daily average;
- **avgWeek(x)**: return the week average for the variable x calculated over all the **avgWeek0(x)** registered in the time. It is the average over the weekly average;
- **avgMonth(x)**: return the month average value for the for the variable x calculated over all the **avgMonth0(x)** registered in the time. It is the average over the monthly average.

For the average valued is present also the percentage operator. The "%" work in a completely different way in respect to the *op* operators. It must be preceded by an *op* symbol and a integer. The complete syntax is "*op d %*" where d goes from -100 to +100. The operator permit to confront two intervals in term of quantity of registered value specific how much the two values must differ. For example, **avgDay(X) >+25% avgDay-1(x)** is valued true if the daily average of x , calculated on day $t-1$, is twenty five percent bigger than the daily average of x calculated on day $t(-1)$. Off course, the % operator, could be used only on comparable data. This doesn't mean it is limited to compare date from the same x . In some case could be useful to confront means from different sensors/resourced. One of them is to compare the **DATA_TRANSMITTED** over **DATA_RECEIVED** in-time.

At first sight, some sensors, might seem unnecessary while other specified in the Snapshot Module are missing. This strictly depends on what the policies should be

able to control. A value like the proximity could be analysed in a policy to understand if the smartphone is, for example, turned face down on a table. In that circumstance, if a call is being made, the behaviour could be considered anomalous. The same thing could be said for the Screen On/off sensor. If the screen is off and the CPU and RAM utilization are very high in-time then a process could be using the resources illicit activities (FlappyBird example).

Two further values are controllable: the maximum value registered over $\text{avgHour0}(x)$, $\text{avgDay0}(x)$, $\text{avgWeek0}(x)$ and $\text{avgMonth0}(x)$:

- $\text{MaxHour}(x)$: the maximum value ever registered for single field from avgHour0 . The value is update only if a new maximum is registered.
- $\text{MaxDay}(x)$: the maximum value ever registered for single field from avgDay0 . The value is update only if a new maximum is registered.
- $\text{MaxWeek}(x)$: the maximum value ever registered for single field from avgWeek0 . The value is update only if a new maximum is registered.
- $\text{MaxMonth}(x)$: the maximum value ever registered for single field from avgMonth0 . The value is update only if a new maximum is registered.

Just to better understand the above keywords, for example MaxHour works in this way: in 24 hours of service are calculated 24 $\text{avgHour0}(x)$, one value for each hour in the current day. If in these 24 average values the maximum number of SMS them MaxHour is equal to that value and will be update in the future if and only if a greater value is registered.

Let see now some example of general policies and why them are important to discover some specific kind of attack. While explaining we will also introduced, case by case, other BePolicy features.

Policy 01: IF $((CPU > 60) \vee (RAM > 60)) \wedge (SCREEN = OFF)$ THEN "There has been a peak in resources utilization while the phone was probably not used".

This policy can discover all these malwares targeting the phone while the user is probably not using it. These malwares, FlappyBird and BadLepricon are two very diffused example, try to pass unnoticed by computing Bitcoin while the user is not using the device and then he will not notice any decay in general performances.

70 is the threshold for CPU and RAM utilization. This general value could be, thanks to tests on the field, also be adjusted to better match with the single user behaviour. If the 70% is too wide or too restrictive for an user, than the policy could be transformed into a user's personal policy and the threshold adjusted for the specific user.

This process could not be done automatically because it depends from user to user. We cannot judge if the behaviour is wanted or unwanted, if the user has installed some specific application and if the user is aware of the phenomenon. We can just observe an anomaly and alert the owner. Do not alert, in case of a real malware, could be considered worse than a false alarm.

A possible solution is to ask to the user a feedback after an alert. If the behaviour is wanted than the threshold value could be raised. But this approach could not

be used with all the service population because requires some skill has be able to discriminate between an evil behaviour and a legit behaviour.

For many reasons, a peak in resource utilization can be caused by an update or a unpredictable system event such an internal control routine.

Policy 01 doesn't not discriminate over time and a single snapshot with the conditions evaluated as true is enough to alert the user.

Then the repeat variable is inserted. This value, inserted after the IF statement, permit to define how many times the policy must be validated before the user is alerted. Three possible controls:

1. Consecutive: permit to define how many consecutive snapshots must validated as true the policy. It is indicated with a (xC) where x is an integer.

- IF (3C) $((CPU > 70) \vee (RAM > 70)) \wedge (SCREEN = OFF)$ THEN
"There has been a peak in resources utilization while the phone was probably not used"

The policy is validated if three consecutive snapshots are evaluated as true.

2. Repeated: permit to define how many snapshots must be validated as true the policy. It works as a counter witch is incremented every time the policy is evaluated as true. It is indicated with a (xR) where x is an integer.

- IF (10R) $((CPU > 70) \vee (RAM > 70)) \wedge (SCREEN = OFF)$ THEN
"There has been a peak in resources utilization while the phone was probably not used"

The policy is validated when ten snapshots over the device lifetime are evaluated as true.

3. Frequently: permit to define how many snapshots over a consecutive set must be validated as true. It is indicated with a $(x - y)$ where x and y are integers. x is the number of occurrences while y is the set size.

- IF(4-10) $((CPU > 70) \vee (RAM > 70)) \wedge (SCREEN = OFF)$ THEN
"There has been a peak in resources utilization while the phone was probably not used"

The policy is validated if four snapshots are evaluated as true inside a set of ten consecutive snapshots.

For this specific subset of policy is necessary to save and keep trace of the counters. Due the fact that the values are personal and relative to the specific user, all the policies with a control over repetition are to be considered personal and then they are copied in the personal user's policies list.

Policy 02: IF $(avgWeek0(SMS_SENT) > +50\%(avgWeek-1(SMS_SENT)))$ THEN "There has been 50% more SMS sent during the last week than the previous one".

This policy states: "if the average number of SMS sent on the week just ended is 50% greater than the SMS sent in the previous week then alert the user". The same

approach could be used for track the calls done over time.

Policy 03: IF ($avgWeek0(CALLS_DONE_SENT) > +50\%(avgWeek - 1(CALLS_DONE_SENT))$) THEN "There has been 50% more calls done during the last week then the previous one".

This policy states: "if the average number of calls done on the week just ended is 50% greater than the calls done in the previous week then alert the user".

Policies 02 and 03 work on mean values but this doesn't introduces delay or further workload during the policies validation. In fact the control over the main value is instantaneous as for punctual value policies. This is possible thanks to the Snapshot Module which compute the main values progressively at snapshot acquisition.

It is also possible set policies to validate application behaviour in time. For each app we know exactly, thanks again to the Module Snapshot, how many bytes have been sent or receive. Than it is possible to set threshold over time over the average sent or received information.

Policy 04: IF ($avgWeek0(DATA_TRANSMITTED_FOR_APP) > +50\%(avgWeek - 1(DATA_TRANSMITTED_FOR_APP))$) THEN "There has been 50% data transmitted during the last week then the previous one".

This policy states: "if the average number of bytes transmitted on the week just ended is 50% greater than the data transmitted in the previous week then alert the user". This policy is general and target all the running application without discrimination. In order to have more control, we introduced the possibility to control the single app by specifying its name inside bracket after the attribute field as in the next policy.

Policy 05: IF ($avgWeek0(DATA_TRANSMITTED_FOR_APP(facebook)) > +50\%(avgWeek - 1(DATA_TRANSMITTED_FOR_APP(facebook)))$) THEN "There has been 50% data transmitted during the last week then the previous one"
This policy states: "if the average number of bytes transmitted on the week just ended by the "facebook" app is 50% greater than the data transmitted in the previous week then alert the user". The control is performed on every app name which contains the "facebook" string.

7.4 Implementation

The Policy Module works on a layer over the Snapshot Module. The server side is built over the snapshots features by adding all the graphic interface useful to define policies and to control, both from the administrator that the user, how these are validated and on which data. After the user has logged in, a specific form permits to him to insert a new policy or edit one already inserted. At the beginning there are not policies inserted. Then the user (or the administrator) can use the dedicated form to insert a new policy as shown in Fig. 7.2.

The insertion procedure is supervised with a series of drop-down menu. First the user selects the sensor to control and then the trigger value. A full policy could be composed of one or more conditions concatenated by using the AND and OR

Figure 7.2: The insertion form

buttons. Last part permit to insert a text description. In the example we inserted the following Policy 01. Any device snapshot sent after the new policy insertion will requires validation. The user can see how his snapshots are validated in the dedicated pages. In this way it is possible to know the device behaviour in time. There validation logs are available for three months and then they are automatically deleted. The same information are available also by a dedicated activity on the client. A three months validation history is a powerful information source because it could permit to understand how many time an event occurred in the past. All the validation and the respective algorithms are performed while the snapshots information are acquired. With the average values already calculated by the Snapshot Module, the only effort and computation are retrieve the policies and the user's values from the database. Then the algorithm merely confront with a series of binary check if the user's value respect the respective policies. Device performance are never influenced. The sensors and other information are already been sent by Snapshot Module.

To understand how overall the Module works let see a complete example (for space constraints we will show just the interest parts of the GUI pages):

1. Administrator inserts manually by the online form the Policy 01 as in Fig. 7.2.
2. User Bob installs the ARAM client and starts to send snapshots as in Fig. 7.3.
3. ARAM server validates every snapshot received. All the snapshots received until now are "normal" and don't validate the administrator policy. Last snapshot validation is shown in Fig. 7.4
4. To simulate a malware background computation we installed Antutu on the Bob client and we started a 5 minutes benchmark test which stress both CPU

and RAM. During the test, with screen turned off, CPU and RAM reach high percentage levels as shown in the last snapshot 10:10:05.0 of Fig. 7.5.

5. ARAM client tries to validate the last received snapshot with the system policies (in this case there is only Policy 01). The CPU is higher than 70, the RAM is higher than 70 and SCREEN is OFF (value 0). The policy is then validated as true as shown in Fig. 7.6. The error is propagated to the client where a new activity is shown as illustrated in Fig. 7.7. Here the user must manually stop the alarm to continue the normal device utilization.

As is it possible to see from the snapshots and validations timestamps, the overall process take less than 4 seconds (including the time to generate the snapshot on the device, send the snapshot over a UMTS network, insert it in the ARAM database and check the policies). User and administrator can control history table for any defined policy as show in Fig. 7.8. For the policy with recurrence in time the behaviour is exactly the same.

Screen	Cpu	Ram	Policy	Result	Data Log
0	13	79.4			2014-11-09 10:04:21.0
1	21	79.0			2014-11-09 10:03:47.0
0	14	76.8			2014-11-09 10:03:14.0
0	15	76.8			2014-11-09 10:02:40.0
1	16	76.8			2014-11-09 10:02:06.0
0	12	81.1			2014-11-09 10:01:33.0
0	14	80.9			2014-11-09 10:00:59.0

Figure 7.3: Bob’s last seven snapshots

This table shows the results of policies setted by the **Administrator**


Policy	Description	Result	Evaluation Date
IF SCREEN=0 AND CPU>60 AND RAM>60.0 THEN FLAG		✓	2014-11-09 10:04:31.0

Figure 7.4: Latest snapshot is validated as normal

Screen	Cpu	Ram	Pr	...	Data Log
0	99	83.0			2014-11-09 10:10:05.0
1	40	84.9			2014-11-09 10:09:46.0
1	18	83.4			2014-11-09 10:09:28.0
1	18	81.5			2014-11-09 10:09:09.0
1	28	81.5			2014-11-09 10:08:51.0
1	80	82.3			2014-11-09 10:08:32.0
0	78	80.4			2014-11-09

Figure 7.5: New snapshot has screen off and CPU and RAM at very high percentage levels

This table shows the results of policies set by the Administrator



Policy	Description	Result	Evaluation Date
IF SCREEN=0 AND CPU>60 AND RAM>60.0 THEN FLAG			2014-11-09 10:10:09.0

Figure 7.6: New snapshot validates the policy

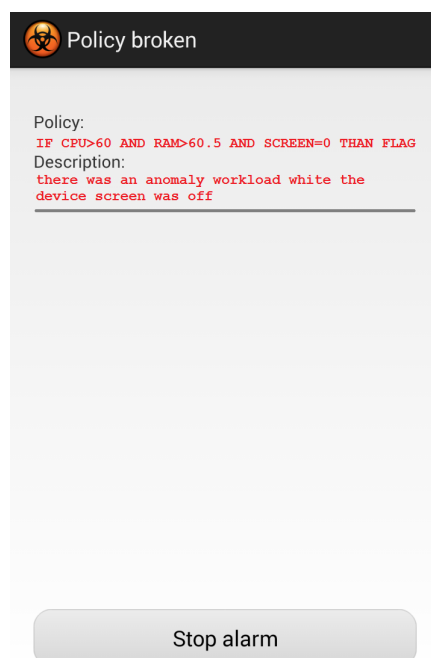


Figure 7.7: Error propagation on the ARAM client








Screen	Cpu	Ram	Pr	s	Data Log	Result
0	99	83.0			2014-11-09 10:10:05.0	
1	40	84.9			2014-11-09 10:09:48.0	
1	16	83.4			2014-11-09 10:09:28.0	
1	18	81.5			2014-11-09 10:09:09.0	
1	26	81.5			2014-11-09 10:08:51.0	
1	80	82.3			2014-11-09 10:08:32.0	
0	70	80.4			2014-11-09	

Figure 7.8: Snapshots validation history page

7.5 Further tests and considerations

Module effectiveness is strictly bound to the policies set. If policies are defined to cover specific malware behaviour or anomaly situation then the system ability to discover both of them will be more effective. Also the possibility to analyse a 3 months history should permit to improve and refine policies to better meet specific case situation. Further, the user can define his own policies and modify the system policies if them are too wide or too limited for his habits.

In the Snapshot Module we presented some malwares developed for testing purpose. Now we will try to see if ad hoc made policies are capable of discovery the specific malwares conduct.

App 24 Analoged Widget has been infected with different piece of malwares. Two of them are related to capability to send SMS and calls to premium number.

The only way to discover such kind of attacks is a comparison with data collected before. If we evaluate a single snapshot (as a series of punctual values) is quite impossible to catch a so high amount of SMS and calls to decide unequivocally that such values are caused by a malware attack. For example, a single snapshot showing 2 calls or 3 SMS since the previous snapshot could belong to a normal behaviour and respect the user's normal habits. It is obviously that the problem could be reduces to pick a valid threshold but again, unless we see a very high amount - as 10 SMS sent in a 15s interval - we cannot decide that 3 SMS are the normal behaviour and 4 are to be considered as an alarming one. Taking into account such considerations, the policies must evaluate the average SMS and calls done in an interval of time such as the day and the week. Policy 02 and Policy 03 cover exactly the behaviour we want to discover. They are capable of discover if the device send more SMS and calls than what has been done in the previous day or week. The 50% is been arbitrary decided but could be adjusted in time by the user if it doesn't represent his habits. Such an approach as the limitation that if the system send 49% more SMS or calls than the previous day/week than it is not detected. For the 24 Analoged Widget the two rules are enough (the malicious influence was very strong and the SMS sent where clearly more than the 50% threshold. We positively used Policy 02 and 02 to discover the attacks after a day. Then we asked our self if the 24 hours discover delay could be

reduced. If the malwares has the same behaviour in time then also an average on hour instead of day could permit to discriminate. Another possible solution could be to compare the SMS and the calls done in an hour with the maximum value ever registered on the device. For example, if the user own the device since 10 months and in the device history the maximum number of SMS sent for hour is 5 then any value greater than 5 could be considered as unusual also if the difference is just of one SMS sent. The new policy would be:

Policy 07: IF ($avgHour0(SMS_SENT) > +5\% (maxHour(SMS_SENT))$) THEN "In the past hour have been sent more SMS than ever"

The policy take also in account a 5% error margin to avoid false positive. The same control can be done over the calls done.

To cover completely the 24 Analoged Widget malicious code and save database query we can use an OR:

Policy 09: IF ($avgHour0(SMS_SENT) > +5\% (maxHour(SMS_SENT))$) OR ($avgHour0(CALL_SENT) > +5\% (maxHour(CALL_SENT))$) THEN "In the past hour have been sent more SMS or CALLs than ever"

The problem with Policy 09 is the impossibility to discriminate the reason for which the policy has been validated as true. In fact when the user gets the alarm message on the device screen he doesn't know if the problem was the SMS or the calls.

Concluding the 24 Analoged Widget malware analysis the best approach to control a malware with this specific behaviour is to check not only the hourly and daily averages but also the maximum data over these interval. In this way the possibility to catch the attack is greater. From our internal test on the malware, we were able to find out the attack with all the four policies due to the massive number of SMS sent. In other circumstance, where the SMS sent are lower and near to the user normal behaviour the possibility to catch the attack is greater if we increases the number of policies which target the value.

BatteryWidget and CurrencyConverter are two other malwares presented in the Snapshot Module. Both of them were infected with a background process designed to steal personal information and to send them to an external server. If the malware behaviour in time is constant it is not possible to find out the sent data by comparing the data sent in different moment. If the malware was sending data in the previous hour and continue to do the same action in the current than the transferred value would be the same. Also the approach to compare the transferred data before and after the application installation doesn't help to discovery a malicious behaviour. The user don't know how many byte the app, in its non malicious version, should generated over time. The only way to track such attack is to use the Application Module and compare the traffic generated by the user app with the traffic generated by the general profile. We will investigate this part in the Application Module chapter.

Chapter 8

ARAM: Application Module

The third module, the Application Module, will be analysed in this section. During the previous chapters, we saw one of the most common attack against the Android ecosystem: the injection attack. Aim of this module is to analyse any application installed, by the user, on the device and compare it to what could be considered the referenced, standard profile of the application. If the installed app differs from the standard profile, which could be normally considered safe, then the anomaly could be considered as a symptom of an injection. In the chapter first we will present the module scope and architecture then we will test its effectiveness against some tampered applications.

8.1 The Idea

Google Play Store is actually, the only and unique, official market place where is possible to download automatically and directly to the phone any Android application. Most of the devices is delivered to the final owner, with this store preinstalled as the default location where to find any desired software, game or utility. Anyway, Google has never prevented the apps diffusion through other channels and the user is free to install an application from developers' website, email direct delivery or any pirate website which offers paid apps for free (cracked applications). While the Google Play store has encountered some security problems in time and what is available from there is not necessarily safe and clean, the probability to suffer from an injection attack is far more probable using an alternative source. If an app does not come from a reputable source then there is an higher chance that the code could be infected or modified to accomplish background malicious activity.

We already saw an example of how it is easy to perform a reverse engineering of an application and then to rebuilt it including a malicious piece of code. With tons of websites offering hacked APKs and no control over installations there is no way for the user to know what he is really installing and using on his device.

The Application Module is design to face such type of attacks. It analyses all the applications the user installs, no matter from where they come, and alert him if any of them differs from the one publicly available from the Google Play Store. On the device, the application information are retrieved by analysing the manifest file but the procedure is not limited to that. Thanks to information acquired by the Snapshot Module it is possible to know other useful information as the traffic each

application generates over the default communication channel.

The overall module architecture is shown in Fig. 8.1. While the acquisition procedure, the communication channel and the server unit dedicated to the storage are similar to the Snapshot module, this new module, to work correctly, needs a to retrieve data from the Google Play Store if necessary.

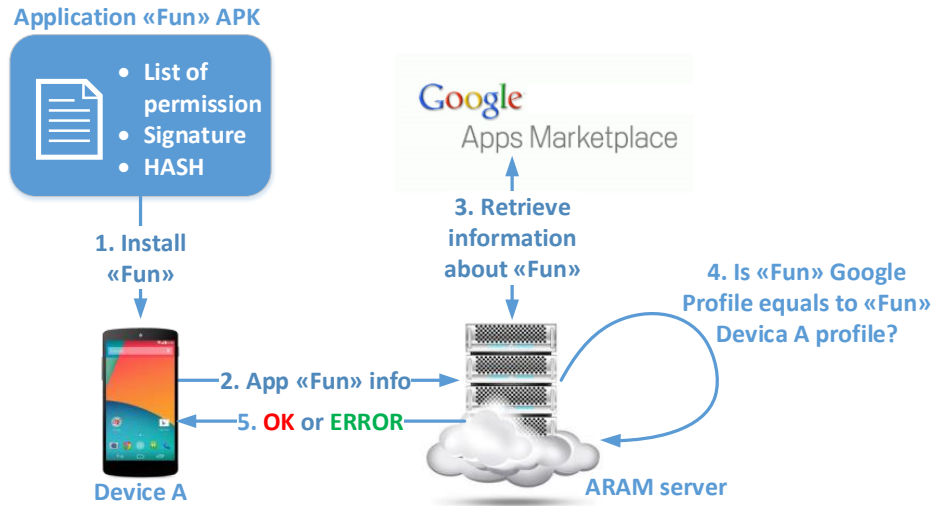


Figure 8.1: Application Module overall architecture

For how Android is developed there is no the possibility to automatically analyse an APK right before its installation. Our goal was to develop a non invasive background procedure and not a tool which requires that user manually checks each single APK file. For these reasons, our analysis begin immediately after an installation procedure has been completed (after the user had agree with permissions and the procedure installation has been accomplished). A dedicated background service intercepts the installation event which follow the installation procedure and, knowing the set of the previously installed applications, understood for difference which app has been added to the device. Then, thanks to the public Android API, it is capable to retrieve a set of information related to the app by analysing its manifest:

- Name: the name of the application;
- Package the name of the package;
- Version: the current version of the application;
- Permissions: the complete list of permissions granted to the app;
- Hash: the hash function of the whole APK;
- Signature: the developer signature over the APK;

These, are saved, as for the snapshot logs, in a SQLite database inside the device storage memory. The structure is quite easy, with one tuple (line) for application. There two further information, not related to what is possible to read directly from

the manifest or Android system: the installation timestamp and the application "status".

Status is a quaternary field which can assume one of the following values:

- Unchecked: the application has been installed and the server has not received or analysed the information;
- Good: the server has received all the information and after an analysis it has concluded that the application is not tampered in any part;
- Evil: the server has received all the information and after an analysis it has concluded that the application has been tampered in some of its parts and should be considered as harmful.
- Unknown: the application is not present on the Google Play Store and then it is not possible to take a decision without further actions;

In the ARAM client graphic interface the above status are shown with coloured icons as shown in Fig. 8.2

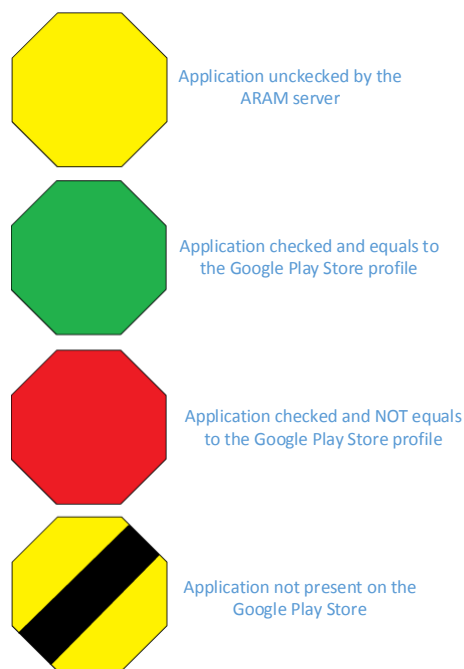


Figure 8.2: Status icons

The communication works exactly as in the Snapshot Module. If an internet connection is available and established, the data is sent in a JSON format over a HTTPS post request. If the connection is not present the data will be sent as soon as one is available. However, the information is never deleted from the internal device database as in the Snapshot Module. This happen in any case, also if the data has been already transmitted correctly and stored on the server side. This decision is easily understandable, the module must be able to understand which applications have already been analysed to discriminate between them on a new installation

event. Further the ARAM client is able to record not only a fresh installation, but also when an app is updated (through internet) or uninstalled (by the user). If any of these events occur then it is registered and the corresponding tuple is updated.

Speaking about the client, Fig. 8.3 shown the Application Module activity at the first execution. How it is possible to see, the module scans the device and adds to its internal database and to the GUI all the applications already installed by the user. All the octagons are yellow because the communication with the ARAM Server is not even started (we turned off the internet connection on purpose to show the GUI progress over time).

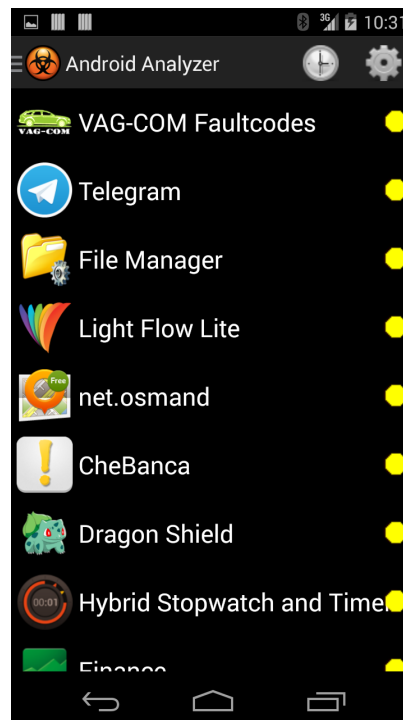


Figure 8.3: Application Module GUI with the applications list

It is fundamental to be able to track also any application update because originally an application could be harmless and become harmful only after getting new code from internet. This is one of the reasons why a static and just a priori analysis is not enough to catch all the infections.

On server side, when the new information is received, two things are done. First, the new application is recorded, with all the respective data, under the user profile. Any user has his own specific application profile with the data recorded on his own device. The table is very similar to the one stored on the mobile device. Then the server checks in its database if the application is already known from a previous hit. If the application is unknown, using the application full name, the server retrieves from the Google Play Store all the information related to it. This procedure is completely free and can be done for any app available on the store without download it. This approach has many advantages in respect to the ones which analyse the APK file locally. First the system could be not effected by the APK presence itself, second it is not possible to download the APK of a non free app without paying for it. With our

approach we can analyse any app without download it. The gathered information are then stored in a database portion dedicated to the general applications profile. These information will be used in the future for any further analysis on the same app version without retrieve them again. Finally the server can compare the user application profile with the general application profile obtained on the Google Play Store. If all the information between the two profiles match exactly then the application has been not tampered while if any voice is different there is an anomaly and probably we have an infected application. The match is a merely string compare among the profiles fields. In both the case, the server send the respective information to the ARAM client which update the "state" value in the application tuple. Then the client GUI is refreshed and the user graphically informed. User has always the possibility to track the applications status by consulting the specific client interface. The same described procedure is performed for any application update. On the server and the device, the user application set of data is conserved until the application is manually uninstalled by the user.

8.1.1 Confrontation Algorithm

The confrontation algorithm over the profiles fields is extremely simple. Any field is treated as an ASCII string and there is a match if and only if there is a complete correspondence between them. It is enough a single discrepancy to have an anomaly. Algorithm can be so summarized:

```
for each field i do {
  if (app_general_profile[i] == app_personal_profile[i])
    continue;
  else tampered_list = tampered_list + app_personal_profile[i];
}
if (tampered_list == empty) app_status = safe;
else app_status = tampered;
```

8.1.2 Applications Profiles

On the server for any application (and for application we consider the pair name-version) there are two complete profiles:

- Personal: represent the specific information related to an application installed on a device. These information are bound to the user and related only to the user. There is a different profile for each user-app pair.
- General: represent the generic information related to the application and it is retrieved by the Google Play Store. There is just a profile for each app.

However, the information available and controllable are also others. Thanks to the Snapshot module we known exactly how many bytes any application has transmitted and received in a specific device. Further at runtime the module computes averages over these values and we know how much data has been generated over a day, a week and a month. Also, the whole recorded data could be used to know how much traffic, in average, any app generates. This is possible by computing the average

over all the users who have installed the same application on their devices. These means values are then associated to the general app profile and are comparable with the each user personal profile by defining appropriate policies in the Policy Module. Accordingly, by combining the data acquired by the Snapshot Module and the Application Module, and defining policies in respect to the general profile, it is possible to discriminate between a normal and an injected app if they differ in the amount of data generated over time.

8.2 Approach Limitations and Heuristic Analysis

The Application Module introduces significant benefits to the malwares discovery and permits to the ARAM architecture to be more effective. However the Module has some limitations and requisites in order to work correctly. First of all, we build the various general profiles by considering apps from the Google Play Store as safe. This assumption is not always true but the Google store is certainly the safest place among all possible sources. This is due because it is the most visible location and then the presence of a malware could be discovered by Google and by the community sooner than any other place.

The profiling, as presented, works only if the original application is available on the Google Play Store. We considered the possibility to build our entire generic app profile by just analysing statistically the data collected from many users. Between all the data gathered, the official app profile could correspond to the app which has more installation among the community. Any app version with less installations than the most diffused, could be considered as a tampered version. This approach present a problem: if a geographic area is strongly targeted by a tampered version and the diffusion of the official untouched and safe release is not common in the rest of the world then the tampered one could be taken as the general profile. However, the statistical approach permits to create a general profile also if the application is not directly present on the Google Play Store (or has been removed because a newer version is present and the user has not yet updated its application).

We then decided to effort this possibility to the users if, and only if, an application is not directly present on the Google Play Store website. This function, called Heuristic Applications Analysis (HAA), must be manually turned on in the client options page (by default HAA is off). User is clearly warned that any analysis and decision taken by HAA is not certain but an estimation over the available data.

In order to implement HAA further information are to be collected:

- Users Counter: for any application is reported the number of users who have the specific version installed on their devices. The counter is decremented when the user remove the app.
- Last Installation Date: for any application is recorded the last installation date.
- Profile: for any application is reported if it is to be considered as the general - and for now safe profile - or one of the possible tampered. Of course this flag could change over time while new information are gathered.

The Users Counter, permits to understand how many users have that specific APK installed. The field is needed to compare among all the other apps having the

same application name but differ for any other field. We have chosen arbitrarily to define a 5%-95% threshold: an application can be considered as the general profile if and only if it has been installed by the 95% of the whole users. If the difference of installation between two applications with same name and version is less than 95% then the module is not able to take a decision and further installations must be achieved. To reduce false positives and limit the changes of decision over small fractions of time, specially at the beginning, we decided also to set a lower bound to the installations number: the HAA system starts to work and assign status only when at least 1000 installations have been recorded. Of course these thresholds could be changed in time to follow the system growth. The profiling is dynamic and could change over time. An application considered safe at a certain time could prove itself as malevolent if more information are collected.

The Last Installation Date is used, in association with the Users Counter, to understand when a specific version is no more available on the market. When the Users Counter turn to 0 (it is decremented each time the user uninstalls the app) no more user has the app installed. Such ongoing could happen in two case: if the application has been update and the older version is no longer available or if it has been removed from the circulation due to security reasons. In both cases it is possible to remove the application and its profile from the database. We decide to remove an application from the database when 30 days have passed since the Users Counter values become equal to zero.

Finally, the Profile field is useful to know at a certain time which profile could be considered as the general and which one not.

To distinguish the normal decision method against the HAA method, we introduced further status and icons as shown in Fig. 8.4. The basic icons are still present and utilized. The newer are used in addition to the normal set because we could have apps validated using the Google Play Store and apps validated with HAA at the same time.

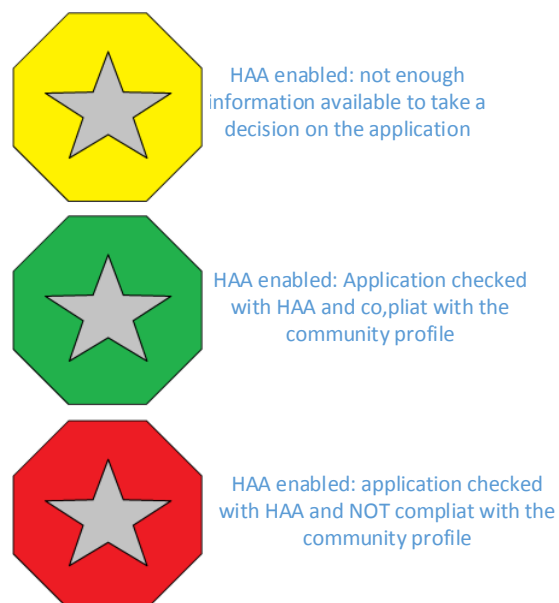


Figure 8.4: HAA icons

8.3 Tests

We will now test our module in two different ways. First, we will analyse some application available both on the Google Play Store and third-party websites. Then we will use HAA to analyse applications which are not available on the Google Play Store and then the alternative analysis has to be used.

8.3.1 Google Play Store

The first set of tests was performed using applications which are available on the Google Play store.

SCENARIO 1: the user, from a third-party website, downloads an APK and installs it on his device. The user doesn't know if the APK is safe or if it is equivalent to the one available on the Google Play Store.

We have chosen as first application Facebook because its APK is easy findable on many free websites. Randomly we download it from [22]. The version available on apk4fun was 20.0.0.25.15, exactly the same one available on the Google Play Store.

Fig. 8.5 shows the Application Module GUI of the client after the application has been manually installed from the APK file. The ARAM server has not been contacted yet (internet communication was turned off on purpose).

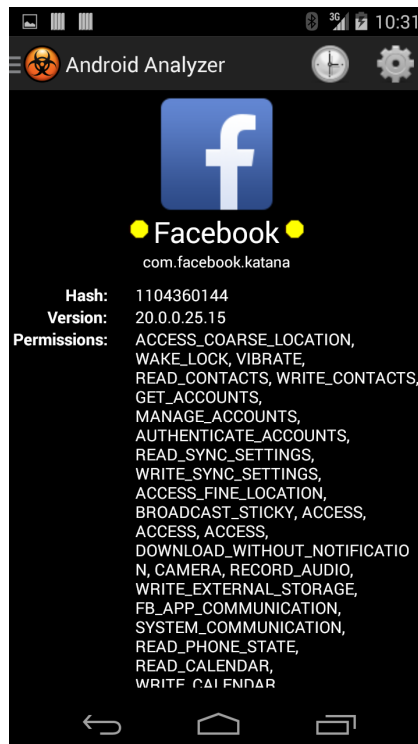


Figure 8.5: Facebook information after the APK installation

Hence we re-establish the internet connection and the Module begins to send data to the ARAM server.

On server side, no available profile is already present for the received data (Name: Facebook - Package: com.facebook.katana - Version: 20.0.0.25.15) and then ARAM searches on the Google Play Store for the Facebook application.

We didn't explain before how this search is carried out. To accomplish the task we used "android-market-api", an open source library freely available on developer webpage [41]. We specifically modified it to integrate its function in our server. With just the application or package full name, this routine queries the Store and finds the Facebook APK among all the available applications and retrieve, without downloading it, all the related fields. The procedure works with free and paid applications. Thanks to these information a general profile is created and inserted in our database. Henceforth it will be use as the general official profile for the Facebook 20.0.0.25.15 APK. Then the ARAM server has all the data needed to check if the user app has the same characteristics as the general profile.

In the specific case, all the information where exactly the same as shown in Fig. 8.6. The confrontation algorithm determines that the application is identical and not tampered.

	Google Play Profile	Unknown APK
Name	Facebook	Facebook
Package	com.facebook.katana	com.facebook.katana
Hash	1104360144	1104360144
Version	20.0.0.25.15	20.0.0.25.15
Permission	ACCESS_COARSE_LOCATION, WAKE_LOCK, VIBRATE, READ_CONTACTS, WRITE_CONTACTS, GET_ACCOUNTS, MANAGE_ACCOUNTS, AUTHENTICATE_ACCOUNTS, READ_SYNC_SETTINGS, WRITE_SYNC_SETTINGS, ACCESS_FINE_LOCATION, BROADCAST_STICKY, ACCESS, ACCESS, ACCESS_DOWNLOAD_WITHOUT_NOTIFICATION, CAMERA, RECORD_AUDIO, WRITE_EXTERNAL_STORAGE, FB_APP_COMMUNICATION, SYSTEM_COMMUNICATION, READ_PHONE_STATE, READ_CALENDAR, WRITE_CALENDAR, MODIFY_AUDIO_SETTINGS, READ_PROFILE, READ_SMS, CHANGE_NETWORK_STATE, CHANGE_WIFI_STATE, READ_GSERVICES, READ_EXTERNAL_STORAGE, CROSS_PROCESS_BROADCAST_MANAGER, BATTERY_STATS, ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE, INSTALL_SHORTCUT, GET_TASKS, RECEIVED_BOOT_COMPLETED, SYSTEM_ALERT_WINDOWS, EXPAND_STATUS_BAR, REORDER_TASKS, CALL_PHONE, SET_WALLPAPER, SET_WALLPAPER_HINTS, ACCESS, INTERNET, READ, WRITE, READ_SETTINGS, UPDATE_SHORTCUT, BROADCAST_BADGES, WRITE_BADGES, 2RECEIVED, C2D_MESSAGE, RECEIVED, RECEIVE_ADM_MESSAGE, RECEIVE.	ACCESS_COARSE_LOCATION, WAKE_LOCK, VIBRATE, READ_CONTACTS, WRITE_CONTACTS, GET_ACCOUNTS, MANAGE_ACCOUNTS, AUTHENTICATE_ACCOUNTS, READ_SYNC_SETTINGS, WRITE_SYNC_SETTINGS, ACCESS_FINE_LOCATION, BROADCAST_STICKY, ACCESS, ACCESS, ACCESS_DOWNLOAD_WITHOUT_NOTIFICATION, CAMERA, RECORD_AUDIO, WRITE_EXTERNAL_STORAGE, FB_APP_COMMUNICATION, SYSTEM_COMMUNICATION, READ_PHONE_STATE, READ_CALENDAR, WRITE_CALENDAR, MODIFY_AUDIO_SETTINGS, READ_PROFILE, READ_SMS, CHANGE_NETWORK_STATE, CHANGE_WIFI_STATE, READ_GSERVICES, READ_EXTERNAL_STORAGE, CROSS_PROCESS_BROADCAST_MANAGER, BATTERY_STATS, ACCESS_NETWORK_STATE, ACCESS_WIFI_STATE, INSTALL_SHORTCUT, GET_TASKS, RECEIVED_BOOT_COMPLETED, SYSTEM_ALERT_WINDOWS, EXPAND_STATUS_BAR, REORDER_TASKS, CALL_PHONE, SET_WALLPAPER, SET_WALLPAPER_HINTS, ACCESS, INTERNET, READ, WRITE, READ_SETTINGS, UPDATE_SHORTCUT, BROADCAST_BADGES, WRITE_BADGES, 2RECEIVED, C2D_MESSAGE, RECEIVED, RECEIVE_ADM_MESSAGE, RECEIVE.
Signature	c412e90e	c412e90e

Figure 8.6: Information related to the two app profiles

Following the result, user's application is marked as safe. The information are forwarded to the device which updates its internal database and the client graphic interface as shown in Fig. 8.7. Here the octagon is turned to green.

The overall procedure requires, on a 10 tests repetition scenario, 1960ms on average, with the device connected by a UTMS network. This could be considered a reasonable amount of time to alert the user, considering also the kind of wireless connection we are using to perform our tests.

By architecture definition, if the user uninstall and then reinstall the same app, the validation procedure should be quicker because the ARAM server already contains the Facebook general profile in its database.

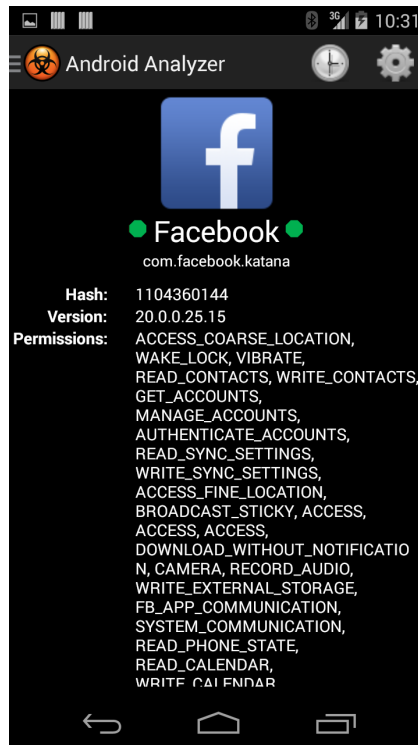


Figure 8.7: Facebook information after the ARAM server validation

We repeated the previous test and, on a 5 tests repetition scenario, we measure an average time of 1440ms. The lower time demonstrate how correctly the module works.

SCENARIO 2: the user, from a third-party website, downloads an APK and installs it on its device. The user doesn't know if the APK is safe and equivalent to the one available on the Google Play Store. We manually modified the APK and inserted non-native code.

The second test we want to perform is similar to the first proposed but in this specific scenario, the user install an application which has been manually tampered by us. The original app is available on the Google Play Store but the user chooses to download it from another unofficial repository. To easy modify the APK, we have chosen an application which source code was freely downloadable from F-Droid (as in the Snapshot Module). In this way we didn't perform a reverse engineering but we edited directly the JAVA source code and recompiled it. The chosen application was AnagramSolver [39] [40]. To temper the application, we merely downloaded it, added a useless line of code (an if condition) and recompiled it. Fig. 8.8 shows the client interface after that the APK has been installed. The icon is yellow to indicate that the app is still unchecked. Then we turned on the internet connection to establish the client-server communication.

Again as before, the ARAM server receives all the user's application information and with name, package and version checks on the Google Play Store for a match. The app is available and then all the relative fields are retrieved and inserted in



Figure 8.8: AnagramSolver after the installation and before the validation

the database (in the app general profile). Then the confrontation algorithms starts and two discrepancies are found: hash and signature values are different and they don't match. We expected such behaviour because we weren't in possess of the developer private key and then we sign the application with a generic private key. The hash value was different due to the "if" condition. The user's application is marked as tampered and the information are forwarded to the client which shows a red icon in the graphic interface and specifics were the discrepancies were found. In red are displayed the general profile information to underline where the information mismatched as shown in Fig. 8.9.

Another test we performed was with Andor's Trail application. Again this app is available both on Google Play Store and F-Droid. This time we add two permissions to the source code before to recompile it. In the specific, we added permissions to read user's contacts (READ_CONTACTS) and to write new contacts (WRITE_CONTACTS). When the user installs this modified app and a general profile is found on the Google Play Store, the discrepancies are easy found and the application is marked as tampered. The information are forwarded to the client which update is GUI as in Fig. 8.10. Here the two wrong permissions are marked in red. Again the signature and the hash mismatches.

Once again, our method was able to discover if a tampered app has been installed.

8.3.2 HAA

The last part to be tested is the heuristic analysis. To the purpose, we wrote ourselves an application called CoverMe. The application is just a proof of concept and doesn't offer a real service to the user. We developed two versions of the application, with exactly the same permissions, but with different behaviour. The difference is only in the code and then the only information in which they differ is the hash value. Both



Figure 8.9: AnagramSolver after the validation

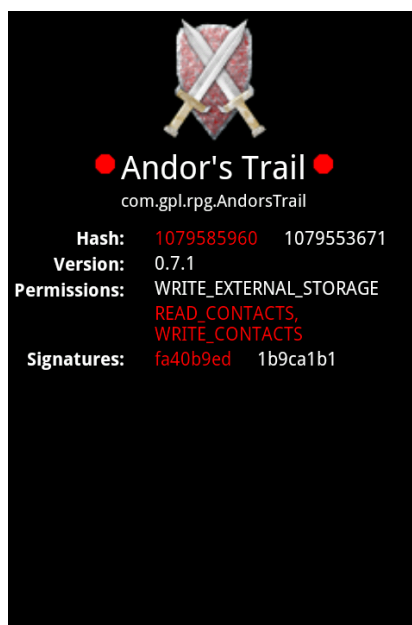


Figure 8.10: Andor's validation permits to discover some discrepancies in the permissions list

have the same signature because we used the same key to sign. We will call these version A and version B. We consider A as the safe version and B as the tampered malicious version. To simulate enough installations we filled directly our database with information about 500 devices. Among these 460 have version A installed while 40 have version B. Then a further user installs the CoverMe application, version B, as shown in Fig. 8.11. The heuristic analysis is off and the internet connection is

unavailable.

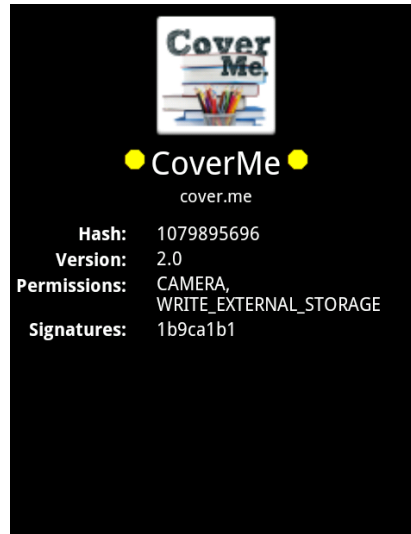


Figure 8.11: Situation after the CoverMe application has been installed

Then, always with HAA off, the internet connection is turned on. The ARAM server receives all the information and tries to find the CoverME app on Google Play Store. The search is unsuccessful and the result is forwarded to the client which updates its graphic interface as in Fig. 8.12. The icon indicate exactly the inability to find the application.

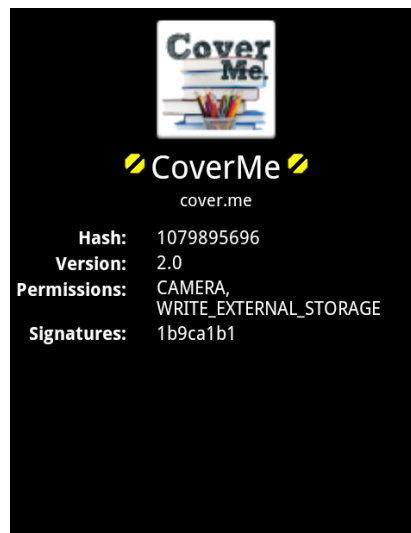


Figure 8.12: The ARAM Server wasn't able to find CoverMe on Google Play Store

Then the user manually turns on the HAA, this is a necessary step if he want a further heuristic analysis. The client sent the information to the ARAM server which is collecting information from the community. Actually there are 40 installations of the same version and the system has not collected yet enough installations to take a decision. The information are forwarded to the client which updates its graphic

interface to show the lack of information as shown in Fig. 8.13.

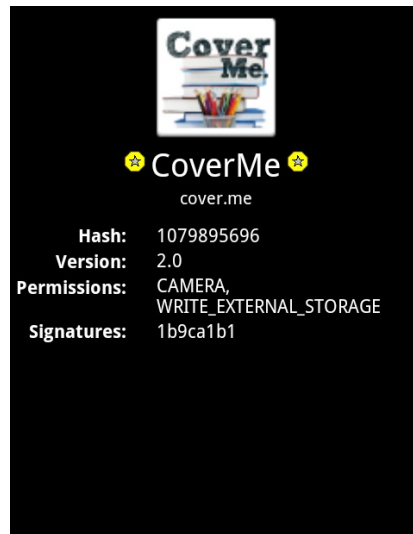


Figure 8.13: The ARAM Server needs more information to take a decision

Then we manually inserted in the database information about the other 500 new devices. Each of them has installed CoverME profile A. Overall now there are 960 installations of profile A and 40 installation of profile B. Finally we have 1000 installations.

At this point, available information are enough to start the computation and the ARAM server classifies application CoverME profile A, as the general profile and CoverME profile B, as the tampered unsafe profile. The information are then forwarded to the client which updates the interface to show the difference between the application installed by the user and the one considered as the safest as shown in Fig. 8.14. The hash value is different and then the information is displayed in red.

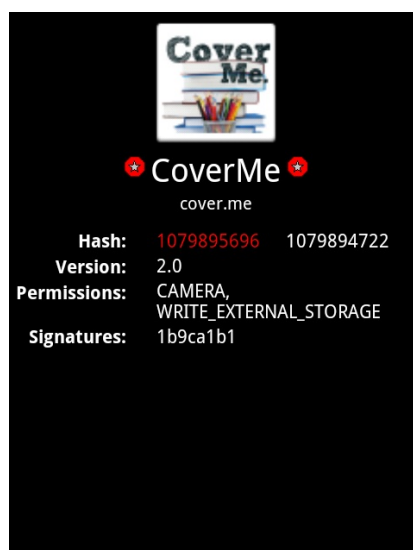


Figure 8.14: The user's app is considered as tampered with HAA enabled

Now we want to show how dynamic the analysis is. We manually edit the database and set the profile B Users Counters to 50000. Then the previous control is no longer valid and now profile B is the most popular and then must be considered as the safest version. The information are forwarded to the client which updates the GUI as in Fig. 8.15.

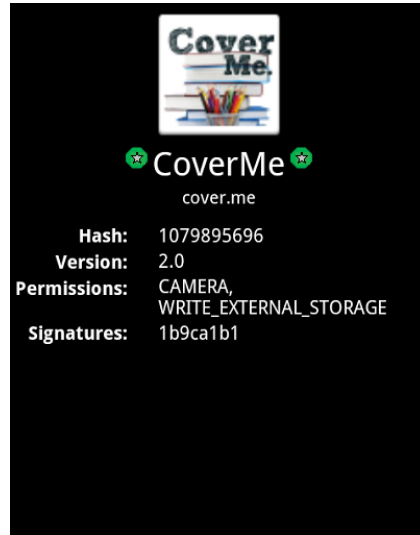


Figure 8.15: The user's app is considered as the safe after more data is gathered

8.3.3 New policies

With the introduction of the Application Module, we know exactly which applications an user has installed on his device. Further it is possible to define generic and personal profile for every app. With all these new information, in association with the Policy Module, it is possible to introduce another class of policies which target the data generated by the application in respect to the personal and general profile. The transmitted and received data for the general profile is calculated doing the mean on all the users which have installed an application which profile matches with the generic profile (the two apps are identical). Due to the effort needed to calculate these means, the function is available only for apps present on the Google Play Store and not for the ones calculated with the HAA function turned on. In the policy definition, the general profile is express using a "*" after application name.

IF ($avgWeek0(DATA_TRANSMITTED_FOR_APP(CoverMe)) > +50\%$ ($avgWeek0(DATA_TRANSMITTED_FOR_APP(CoverMe*))$) THEN "During the last week, the installed application has transmitted 50% more data than its respective general profile".

This policy states: "if the average number of bytes transmitted by the target application has outdated, on the week just ended, by 50% the data transmitted by the respective general profile then alert the user". In this manner it is possible to know if an application transmits in average more or less data then its respective general profile.

The same policy could be used to discover the BatteryWidget and CurrencyConverter malwares presented before. For example we can compare the hourly or daily averages data generated by the apps with the maximum average ever registered by the general profile:

IF (*avgHour0*(*DATA_TRANSMITTED_FOR_APP*(*BatteryWidget*)) > (*MaxHour*(*DATA_TRANSMITTED_FOR_APP*(*BatteryWidget**))) THEN "During the last hour, the installed application has generated more data than its respective general profile".

The above policy is able to understand if the app transfers more data than its general profile has ever transferred in time. For example, if the general profile has generated in an hour a maximum of 2,56Mbyte of data and the user application in the same period has transferred more data than the general behaviour should be suspect.

To prove such thesis we repeated the test done for CoverMe but with the CurrencyConverter application. We installed on five devices five copies of CurrencyConverter. Four of them were clean, as present on the Google Play Store, while one was infected with a malware which collects information from the device and sends them to an external server. We didn't know on which device the tampered version has been installed. CurrencyConvert, in both the versions, generates periodically a request to a remote server in order to get currency rates updated. Normally the traffic generated is really small. We were not interested in the other application information and we evaluated only the traffic generated. Off course the application signature and hash were different in the infected version (in respect to the Google Profile).

In a 12 hours test, the ARAM server registered the values shown in Fig. 8.16. The transferred data is measured in bytes.

	Device A	Device B	Device C	Device D	Device E	General
recording time	12h	12h	12h	12h	12h	
interval	30s	30s	60s	60s	300s	
avgHour	717	721	720	1989	719	720
maxHour	721	721	720	2022	722	722

Figure 8.16: Traffic generated by the application in different devices

The application installed on the device D is clearly the infected because it generated a complete different amount of data over time. Both AvgHour and maxHour differ significantly from the general profile built analysing the information collected from the installations. Then the malicious behaviour could be discovered by policies targeting the AvgHour information and the MaxHour information.

Chapter 9

ARAM: Identity Reinforcement Module

The fourth module, the Identity Reinforcement Module, is designed to reinforce the user identity when an action, performed via the mobile device platform, is carried out on any web site which requires a login process and then where information and actions are strictly personal. Web sites such as Facebook and Twitter are widely used by the internet community and the smartphones diffusion has shift the way in which people access to these contents. In fact, smartphone owners are constantly connected to internet and most of the actions which were done before on laptops and desktop now are performed thought the mobile devices.

According to an IDC study [43], among smartphone owners on age 18-44, 84% of time is spent interacting with email and social networks. Overall, 7 over 10 smartphone owners access Facebook via their mobile device and these users spent 132 minutes each day using their device. Of that total, Facebook accounts for 25% - about 33 minutes. Of that 33 minutes spent connected with Facebook via mobile, 16 minutes are spent browsing news feed, nearly 10 minutes messaging and just over 6 posting status updates and photos.

In front of these data, it is fundamental to protect the user and let him act in completely safety. The purpose of the first three modules we proposed was to discover any potential harmful application analysing the information acquired directly on the mobile device. These methods don't protect the user identity among internet ecosystem. Cases in which, for a serious of reasons, a third-party is able to seize the user password or the user identity are not prevented.

In this chapter we will see how it is possible to use the ARAM's data to control the identity of who is performing an action on remote websites and hot it is possible to deny the request if the requester information does not match with the ARAM user profile.

9.1 The Idea

Any user who installs the ARAM client is constantly feeding the ARAM server with personal information. While some of them are not really personal and the same values could be seen in different user's logs, other are strictly personal. They become far more personal if they are seen as a set and analysed together. In this way we are

building a user profile. This profile should not be viewed as a way to discriminate the user among other available profiles but as a way to control the user behaviour in time and discover anomalies in respect to what has been recorded. The module architecture is shown in Fig. 9.1.

Again, the method starts with the Snapshot Module which collects device snapshots in time and perform its usual computations (database storage and averages). These information are already available on server side to be used for all the analysis we saw in the previous chapters. Then no further computational resources are requested.

Afterwards the user, already registered and logged into an online website as could be Facebook (just to follow the architecture example), wants to perform some kind of action related to the specific website nature. All the interactions user-website are performed through the dedicated Facebook client for Android devices. Here begin the new module operations.

With the user's request - for example to post a new message on his public wall - the Facebook client need to send also a snapshot of the mobile device computed at the request time. To do that and to take advantage of our architecture, the Facebook client needs to be modified in order to meet our protocol. Any available official client could be easy modified because it has just to call a specific function of the Snapshot Module to get a valid snapshot.

Any request performed to the server should have this specific syntax:

[request ; snapshot]

Where request is the usual way the client interacts with the remote server (usually by calling a specific API through a proprietary library as example) and snapshot is a collection of sensors values and resources calculated on the devices exactly before to deliver the request message.

The Facebook server receives the message and do not perform as usually. It frozen the user's request and contacts the ARAM server forwarding the user's identity and the snapshot.

The ARAM server controls if the received snapshot belong to the user profile. The answer is returned to the Facebook server. If the control has been positive (the snapshot is congruous with the user profile) then Facebook performs the request and adds the new message on user's wall. If the control has been negative (snapshots is not congruous with the user profile) the Facebook server:

- doesn't perform the request;
- alerts the account owner by email;
- log-out the user and insert the mobile device in the blacklist;

If Facebook has negated the request while it was legitimate, the owner can follow a link in the email and he can insert a special password defined during the registration. This password is not stored on the mobile device and it has been never used before for the login procedure.

9.1.1 Requisites and Constraints

The Identity Reinforcement Module, to work correctly, need the collaboration of the server where we want to reinforce the user identity. In fact the server, Facebook in the



Figure 9.1: Identity Module Architecture

example, is an active part of the architecture and it needs to handle messages which follow the ARAM protocol. We don't believe that this is a too strong constraint because the required changes are minimal. Most of these websites offer a dedicated client, distributed as an Android APK and available on the Google Play Store. These clients could be easily modified to support the ARAM architecture. The second requisite is a direct consequence of the first. The modified client must be the only way the user has to interact with the website. If an attacker or the users himself is able to access the site via a common browser than all the reinforcements are easily bypassed. The third requisite is quite obvious: the user has installed the ARAM client on his device (ideally at the device first power on) and he has always sent correctly the snapshots to the ARAM server. The fourth requisite is the presence of an internet connection but this a minor problem. Indeed if the user wants to access the Facebook website and post a message he needs a working connection anyway. Fifth, the approach is meant to prevent identity theft episodes and not to offer a protection if the mobile device is stolen and the attacker uses the legitimate smartphone to perform the requests. Finally, the ARAM server is a single point of failure. If it is compromised or replaced by another server all the security architecture falls.

9.2 User Profile and Confrontation Algorithm

Among the information gathered by the Snapshot Module some are relevant for user identity reinforcements while other are not. Now we will now analyse each attribute to understand if it can contribute to the overall identification process.

- IMEI: the number is unique among all the devices and it is used server side to identify who is sending the information. The IMEI number in the request must match the one which belong to the user's device. Still this number could be

forged easily. If the device is used then the previous owner knows it. Further, any application, with the right permission (`READ_PHONE_STATE`) can read it and forward the information to a remote server (and then be used by someone else in a future attack). To end up, IMEI has its importance but it is too easy to steal and replicate.

- **SCREEN**: know if the screen is on or off do not add any useful information. Off course, if the user perform manually an action then the screen must be on. This is the only check this field can offer. Anyway the information is useless because any tampered request would has screen field set to on.
- **CPU**: know the CPU workload could be representative if we analyse the CPU behaviour in an interval of time. With a single snapshot and only a punctual value available, the CPU information is useless to perform any control.
- **RAM**: exactly as the CPU, a single snapshot doesn't offer any valuable information. Also if we consider the RAM and CPU together they are asymptomatic because these two resources are not directly related, especially in a system with virtual machines and shared among other applications. It could be possible to define thresholds among the values analysing the user's behaviour in time but again, a punctual value is too fluctuating. The single snapshot could be affected by any sudden internal Android routine.
- **PROCESSES**: the running processes are a very personal information. While the whole list is long and complex, if the operating system related services are removed, what remains is a set of very few applications installed by the user himself. This set is personal and normally it varies little over time. Further, thanks to the Application Module, we know exactly each application that the user has installed or removed from his device. Then the list contained in the snapshot could be checked against the user's applications list in the ARAM server. The user's applications list must exactly matches the snapshot content, any difference is symptom of a problem. There is not the possibility of a synchronization problem. If for some reason, the internet communication is missing and the user installs or removes an application, then the first time the internet is restored all the modifications are propagated to the ARAM server. Then, when the user performs the request to the Facebook server, the list on the ARAM database is up to date. If for some reason the ARAM server is not up to date - because the user has stopped the ARAM Snapshot Module or because there is a connection problem related only to the device-ARAM link - the timestamp inside the user's request will be bigger (older) than the last registered snapshot on ARAM server. In this circumstance our architecture is not able to guarantee a perfect security check. It is up to the administrator to decide to refuse the Facebook request or to wait for all the data to be correctly propagated.
- **DISK**: the free internal space available is a useful information. This value is really related to the device in use. Any request should have a value compatible with the last snapshot recorded on ARAM server. If the ARAM server is up to date, the value in the request should match exactly with the database or differ

only by a small amount. Indeed, knowing the user profile and all the average values it is possible to have a good estimation over the upper bound. However any estimations become less reliable as time passes between the last snapshot recorded on ARAM server and the one contained in the user's request.

- **PROXIMITY**: the distance between the device and the first object in direct line is useless because it can vary drastically in a fraction of time.
- **LIGHT**: the quantity of ambient light is almost useless because is not something we can profile and the behaviour is unpredictable in time. It is enough to turn on a lamp or change room to distort the measured value.
- **ACCELLEROMETER**: the acceleration force could be a useful information if analysed in an interval of time but with a single snapshot it represent nothing and it is too influence by a single movement to be profiled. For example, the value is completely different if the user is moving the arm rather than holding it while using the device.
- **BATTERY LEVEL**: the percentage of battery charge remaining is useful and could work as a threshold. If the internet connection is available then the value must match. The value is influenced if the device is connected to the power during two different snapshots and the second has not been recorded yet. With the `Battery_Alimentation` field we can track these events but normally during two conservative snapshots the value changes a maximum of one or two percentage points.
- **POWER SOURCE**: the kind of power source the device is using (battery, USB or wall). This information could be used as an equal or not equal check. But the user could change the device power source between two consecutive snapshots and then the value registered in the ARAM server may differ from the request. Thus, the information is useless.
- **BATTERY TEMPERATURE**: the Celsius degree measured on battery sensor is not very useful because the environment can influence too much a single snapshot. For example, a sunbeam from a window can raise the device temperature of 10-15 degrees in a few seconds.
- **NETWORK**: the channel used to connect to internet (WIFI or 3G/LTE). Again this value is useful to build a profile in time while the single snapshot can deceive. If the user loses the 3G connection for some seconds and in consequence the device switches automatically to the WIFI then the single snapshot value doesn't represent the real behaviour.
- **DATA RECEIVED**: The total bytes received since the device was turned on is a really personal and discriminant information. The value works as a counter and then each request must contain a value bigger or equal to the ARAM last recorded snapshot. The value is very difficult to falsify and the attacker should know when the user is usually to turn off the device (the counter is reset each reboot). If the internet connection is missing for a small period of time this value is relatively little affected because it should be just equal or bigger. It

is also possible to use the user profile to calculate a threshold if internet with the ARAM server is missing or the request is sent between two consecutive snapshots. For example, the user has received 124MByte of data in the last recorded snapshot and the interval between two consecutive snapshots is set to 5 minutes. If the user downloads a 2 MByte photo and then he sends a Facebook request, the ARAM server is not aware of the new transfer if 5 minutes from the last snapshot are not elapsed. When Facebook forwards the request, the control ARAM does are two: first it checks if the received MBytes are equal or greater to what has been recorded - 126MByte are greater than 124Mbyte - and second if the amount is congruous with the user average received data between two consecutive snapshots - with an average of 1,2MBytes there is an incongruence because the device has transferred more MBytes then the last recorded snapshot plus the profile average (124Mbytes + 1.2MBytes). A solution is to use the maximum received data recorded between two snapshots, an information already recorded by the Snapshot Module. With the maximum received information we have a dynamic threshold to know how far the value in the request can be greater than the last recorded snapshot. This check is so calculated:

$$a \leq b \leq a + c * p$$

where a is the last available data on ARAM server, b is value contained in the request, c is the maximum received data between two consecutive snapshots and p is a coefficient equal to 1.25 introduced to limit the false positives cases.

- **DATA RECEIVED FOR APPS:** the better way to handle this information is with an equal or bigger check and use the app profile to stablish a threshold. For each app we know, thanks to the Snapshot Module and the Application Mobile, how many bytes have been received in average (snapshot, hourly, daily, weekly and monthly) and the maximum data transferred between two consecutive snapshots. Again we use the maximum amount of data received between two consecutive snapshots as a threshold:

$$a \leq b \leq a + c * p$$

where a is the recorded received data on ARAM server for each application the user has installed, b is the value contained in the request, c is the maximum received data between two consecutive snapshots for the single app and p is a coefficient equal to 1.25 introduced to limit the false positives cases. Contrariwise, if an application has not received data then the check must exactly match.

- **DATA TRANSMITTED:** exactly as DATA RECEIVED. It is possible to use the maximum transmitted data between two consecutive snapshots to define a threshold:

$$a \leq b \leq a + c * p$$

where a is the recorded transmitted data on ARAM server, b is the value contained in the request, c is the maximum transmitted data between two consecutive snapshots and p is a coefficient equal to 1.25 introduced to limit the false positives cases.

- DATA TRANSMITTED FOR APPS: again as before, the better way to handle this information is with an equal or bigger check and use the app profile to establish a threshold. For each app we know, thanks to the Snapshot Module and the Application Mobile, how many bytes have been transmitted in average (snapshot, hourly, daily, weekly and monthly) and the maximum amount of data transferred between two consecutive snapshots. Again we use the maximum transmitted data between two consecutive snapshots as a threshold:

$$a \leq b \leq a + c * p$$

where a is the recorded transmitted data on ARAM server for each application the user has installed, b is the value contained in the request, c is the maximum transmitted data between two snapshots for the single app and p is a coefficient equal to 1.25 introduced to limit the false positives cases. Contrariwise, if an application has not transmitted data then the check must exactly match.

- CALLS: the field contains the number of calls done, received and missed since the device has been turned on. Working as three counters, the values in the request are used as an equal or bigger check against the three contained in the ARAM server. For these values don't exist any average between two snapshots on ARAM server. The reason is quite simple, the number of calls during two consecutive snapshots are unpredictable and then we decided to define arbitrary thresholds. The checks are so performed:

$$a \leq b \leq a + c$$

where a is the recorded calls on ARAM server, b is the value contained in the request and c is the threshold so defined:

- 1 if the interval is set to 15 seconds;
- 2 if the interval is set to 30 seconds;
- 3 if the interval is set to 60 seconds;
- 5 if the interval is set to 300 seconds;

- SMS: the number of SMS received and sent. The values are two counters. Again we decide to use arbitrary thresholds instead of averages. Further the received value between two snapshots doesn't depend strictly by the user behaviour but it is influenced by external parts which may send SMS randomly. The checks are so performed:

$$a \leq b \leq a + c$$

where a is the recorded SMS on ARAM server, b is the value contained in the request and c is the threshold so defined:

- 2 if the interval is set to 15 seconds and the field is SMS_sent;
 - 3 if the interval is set to 15 seconds and the field is SMS_received;
 - 3 if the interval is set to 30 seconds and the field is SMS_sent;
 - 4 if the interval is set to 30 seconds and the field is SMS_received;
 - 4 if the interval is set to 60 seconds and the field is SMS_sent;
 - 5 if the interval is set to 60 seconds and the field is SMS_received;
 - 9 if the interval is set to 300 seconds and the field is SMS_sent;
 - 10 if the interval is set to 300 seconds and the field is SMS_received;
- GPS: the user coordinates are a very personal and discriminant information. The control could be performed by calculating the distance between the last user location (recorded on ARAM) and the new location (contained in the request). The formula we used to calculate the distance between the two terrestrial points is the spherical law of cosines because is one of the easiest to calculate and has an average error of just 0.3%. It approximates the geode to a sphere ($R = 6372000,795477598$ m). Given two point A and B with these syntaxes:

$A = (minlon, minlat)$ where minlon and minlat are minimum longitude and latitude expressed in radians

$B = (maxlon, maxlat)$ when maxlon and maxlat are maximum longitude and latitude expressed in radians

The distance between A and B is calculated as:

$$distanceAB = arccos(sin(minlat) * sin(maxlat) + cos(minlat) * cos(maxlat) * cos(maxlon - minlon)) * R$$

The distance is expressed in linear meters. Again we choose to define T as arbitrarily:

- 1000m if the interval is set to 15 seconds;
- 2000m if the interval is set to 30 seconds;
- 5000m if the interval is set to 60 seconds;
- 30000m if the interval is set to 300 seconds;

Finally distanceAB must be inside the range:

$$0 \leq distanceAB \leq T$$

If distanceAB is not included in the above range then there is an anomaly.

- **PRESSURE:** the environment pressure could be a very useful information but, again, just if it is analysed in a period of time. The punctual value is too easily influence by single ambient events. Then, the control should just check if the value complies to the last recorded snapshot. We decided arbitrarily to define a range where the request data should belong to:

$$(a/100) * 95 \leq b \leq (a/100) * 105$$

where a is the recorded pressure (ARAM server) and b is the value contained in the request. The ten percentage range has been decided after a test on the field. We asked five users to install the client on their smartphones and we recorded snapshots for a week. The recorded pressure for one of these client is shown in Fig. 9.2. The users shared the same behaviour then we will analyse just one of them. In the specific case, the average pressure was of 976 mbar with a lowest value of 964 (1%) and a higher value of 989 (2%). Following the real case, a massive variation is not possible between two consecutive snapshots.

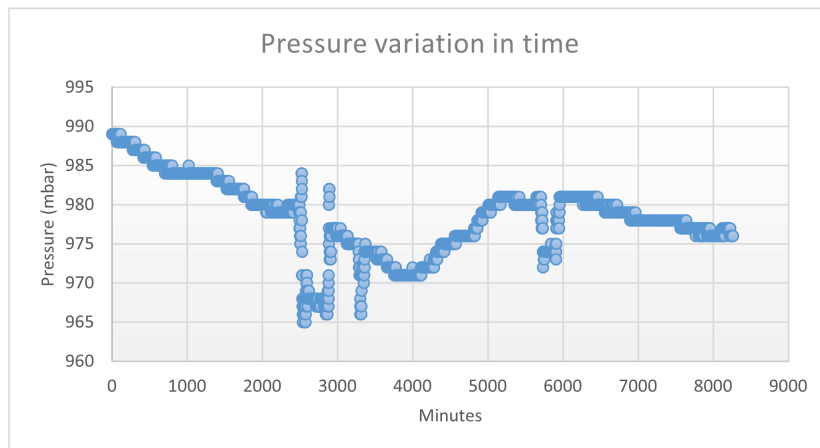


Figure 9.2: Pressure Values Reordered on a Client

- **STEPS:** steps done in a day are a discriminant information but how these steps are registered limit their trustworthiness. The measurement is performed using the accelerometer and other device information and then it is just an approximation. Our internal tests showed that the sensor can measure a step even if the user is sitting in a chair and he is moving an arm. With the introduction of the wearable devices, with specific new fitness sensors, the steps acquisition should be more accurate and realistic in future. Further Android 5.0 should introduce new API related to the user movements. The contributes of these new device could decisive for the module adoption. With an accurate measurement it could be possible to understand user's habits. For example an user could walk every morning two kilometres to reach his workplace. If we trace the user during a working day and the steps count doesn't match with the steps he usually does then there is an anomaly. Further with time slots division (morning, afternoon, evening, night) it could be possible to know exactly

how many steps the user does in precise periods of the day and then perform finer controls. Due the above considerations, for now, we decided to use the sensor just as a counter and perform an equal or bigger control. In future revisions of the architecture will be possible to understand if the new chips and API will introduce relevant corrections and improvements.

- **TIMESTAMP:** timestamp is a powerful information because can tell to us exactly how many seconds or minutes are elapsed between the last recorded data and the request. For any other purposes is useless since it is equal to every device: it measures the elapsed time since the first Unix machine has been turned on.

The control procedure and the above explained thresholds are taken to the extreme only with very large snapshots interval acquisition. The user should select large intervals - 5 minutes for example - only if he has strong constraints over battery or traffic. We recommend, and it is clearly stated in the client interface, to use 30s or 1m interval. 15s and 5m intervals are present only for specific exceptional circumstances.

The overall algorithm used in the control procedure exactly reflects what said above. All the chosen fields - those whom add significant and discriminating information - are checked one by one sequentially. If one of them is not congruous to the specific defined in the analysis then the procedure is stopped and the request is rejected.

9.3 Design and implementation

With the Identity Reinforce Module we offer a service to any online web site which want to do a further control on its users in time. Off course, to develop, implement and test the module we could not count on a collaboration with any famous web site as Facebook, Twitter, etc. While the ARAM server was already in place and working for the other modules, to test the new procedure, we needed an online actor who interprets the Facebook server as in our architecture example. To the purpose we develop a specific server and relative web page. The web page works as an public bulletin-board. Here, all the registered users can post and read other users messages sent by mobile devices. To access this online bulletin-board, the user installs on his device a dedicated client distributed as a normal APK. Its interface is quite simple: there is just a text field which permits to insert an ASCII string and a SEND button to append the new message to the online bulletin-board. When the user pushes the SEND button, the client sends an intent to the Snapshot Module which acquires a new snapshot and then forwards it to the bulletin-board client as a whole object (a JSON object). The client sends the ASCII message and the snapshots object to the bulletin-board server. The bulletin-board forward the snapshot object to a special socket on the ARAM server and wait for a response. The ARAM server performs the checks on the snapshot (with the thresholds explained before) and forward an "OK" or "ERROR" token to the bulletin-board. If the answer is "OK" then the message is added otherwise it is discarded. All the communications are over a secure HTTPS connection. ARAM uses the IMEI in the snapshot object to retrieve the

correct profile. Our proof of concept design is extremely simple but enough to test the new module accuracy.

9.4 Tests

We identified two possible tests to try the Identity Reinforce Module:

1. legitimate snapshot: along the request is sent a valid snapshot (generated by the right device);
2. tampered snapshot: along the request is sent a tampered snapshot (generated by the right device but that has been manually modified);

We didn't schedule a test that might seem essential: clone the device. In our circumstance this could be seen as to copy the original IMEI into another smartphone and to generate requests and snapshots with fake device. This specific attack belongs to the tampered snapshot case and it is discoverable with the exactly same control. Further the attacker must be able to guess the right Facebook Username and Password to send valid requests.

Another question to investigate is how the explained architecture works against the man in the middle attack, an entity positioned between the client and the Facebook server which is able to listen and store all the information transiting on the communication channel.

First of all, all the communication are protected thanks to the HTTPS protocol. The security of HTTPS is based on the underlying TLS, which uses long-term public and secret keys to exchange a short term session key. This last is used to encrypt the data flow between client and server. Then, a man in the middle is unable to decrypt the information on the channel without the proper secret key.

Second, an attacker needs also to know the user login and password information to connect correctly to the service. If these information are stolen then we belong again in the clone attack behaviour and the second tranche of tests.

9.4.1 Legitimate Snapshots

As many time before, we tested the module using five different devices, each one belonging to a different user with different habits. On each smartphone has been installed a modified version of the BulletinBoard client. This version is exactly as could be any web site client but with an automatic message dispatch in time to not bother the user and have the same number of sent messages for device.

The test was so performed:

1. each user installs the ARAM and Bulletin-board clients on his device;
2. each user runs the Snapshots Module for a week (in order to have daily and weekly average values). The interval is set to 30 seconds for the first two devices, 60 seconds for other 2 devices and 300 seconds for last.
3. after a week the each user starts the Bulletin-board application. Each client sends automatically a message to the Bulletin-board server every 15 minutes

with a random delay of +/-2 minutes. This is done to have different delays between requests and snapshots. These messages are sent for 72h while the user operates normally as usual. The only condition imposed to them was to not turn off the device while the tests were carried on.

4. at the end of the 72h we recorded how many requests have been validate as "OK" and how many as "ERROR".

Fig. 9.3 shows the test results. How could it is possible to see, each request from Device A (137), Device B (146), Device C (141), Device D (152) and device E (139) have been successful validated (the snapshot contained in the request matched with the user last snapshot and profile). The only exceptions could be observed on Device A where two requests were refused and their respective snapshot marked as invalid. We didn't aspect such behaviour and then we perform further controls. The problem was quickly discovered: both the requests contained a 0 value in the pressure field while the user profile had an average value of 911. To understand the cause we did further tests and the phenomenon was repeated and other 0 recorded. We concluded that the pressure sensor presents a malfunction behaviour, something not related to our architecture but to the sensor electronic itself. Such malfunction could not be predicted from our architecture. The system was unable to discriminate between a tampered value and an hardware malfunction. Any way such a behaviour - have a random electronic fault in a sensor - could be seen as a very personal information. Only that specific device suffers of that problem and then, with a sufficient set of snapshots, it could be registered in the user profile. We didn't investigate this possibility and will study it in future works.

	Device A	Device B	Device C	Device D	Device E
recording time	72h	72h	72h	72h	72h
interval	30s	30s	60s	60s	300s
total requests	137	146	141	152	139
valid requests	135	146	141	152	139
invalid requests	2	0	0	0	0
average delay	4124ms	3758ms	4467ms	3122ms	3989ms

Figure 9.3: Requests Valuation Response

Last table line displays the average time to complete a full architecture step (from when the device sends the request to when it receives the final response). The first hoop (device to bulletin-board connection and vice versa) was performed over a UMTS network while the board and the server where linked by cable (LAN 10/100). In any case, the two 3G communication (from and to the device) took most of the time. This depends on the quality of the signal and where the device is positioned. Overall the times are acceptable for the field of use. We must also consider that the action to be performed - post a comment - requires less time than the overall architecture time because it is computes before the last hop.

9.4.2 tampered Snapshots

We repeated the tests with the same schema but with a difference. Any snapshot attached to the request contains a tampered value which belong to one of the other four devices (randomly). For example, Device B calls was replaced with Device A value and Facebook transmitted data from Device C has been inserted in Device D and so on. The fields are switched directly on the ARAM server before perform the controls. We used such an artifice to facilitate the our task. Test results do not dependent from where the data were modified.

	Device A	Device B	Device C	Device D	Device E
recording time	72h	72h	72h	72h	72h
interval	30s	30s	60s	60s	300s
total requests	134	141	143	167	128
valid requests	0	0	0	0	0
invalid requests	134	141	143	167	128
avarage delay	3929ms	4051ms	3498ms	3452ms	3764ms

Figure 9.4: Requests Valuation Response

Exactly as expected, Fig. 9.4 shows that is sufficient to alternate a single field to invalidate the profile recognition. The sensors chosen for the procedure and the thresholds fixed then permit to easily discover any relevant variation in the user normal behaviour. This demonstrate how the proposed approach is valid and how it is possible to reinforce the controls over the user identity without disturbing the user and without changing his habits.

Chapter 10

Further Data Mining Studies

During the ARAM architecture overview we analysed and checked the acquired data - the snapshots - with statistical approaches. We defined policies and thanks to the Snapshot Module information we controlled the device behaviour in time. To complete our work and offer an alternative data mining method we will now investigate the use of an the WEKA libraries [56]. Weka is a collection of algorithms for data mining analysis wrote in JAVA and freely available. It contains tools for data processing, classification, regression, clustering, association rules and visualization. We will use these libraries as a "blackbox", a common approach in the development ecosystem. We are not interested in how the library's algorithms are implemented or how the mathematical aspects were addressed. We want just to perform tests on our snapshots database using the tools at the highest possible level. Final aim of the chapter is to understand if WEKA can offer superior results compared to our statistical approaches. For this we will compare WEKA results with those obtained by our architecture.

10.1 Identity Reinforcement

The first point we want to investigate is if it is possible to understand, after a specific user profile has been generated, if a single new snapshot - never saw before - belongs, with a reasonable certainty, to the profile owner or not. We don't want to be able to understand from an anonymous snapshot who is its owner. We know exactly from which device it has been generated. What we are trying do to is exactly what the Identity Reinforce Module does but with a supervised learning classification algorithms. It tries to recognize the user behaviour from a sequence of his snapshots. Then after a profile has been calculated, we will see if a new and never saw before snapshot may fall in the estimated profile. To do the tests, we collected 72 hours of snapshots from two different smartphones (belonging to two different users with different habits). We will call these two sets of snapshots as A and B. What we want to do is to use a subset of A to train a supervised learning algorithm. Then we will use random snapshots from the rest of A and B to test our classifier and see if it able to discriminate if a snapshot belong to A or it comes from an unknown entity (B in the case). We can't train our system with B because B is just a possible source of snapshots. Our system should be able to discriminate between A and an infinite possible source of other users' snapshots.

With a 15 seconds interval we collected 17280 snapshots for A and the same amount for B.

First step was to transform every snapshots into a feature vector, which contains a number of features that are descriptive of the object, in our case each sensor or registered values was considered as a features.

Second step was to determine the learning algorithm. Our problem belong to the unary classification problem which tries to identify objects of a known class amongst all the available objects by learning from a training set containing only objects from the known class. The task is far difficult than distinguish between two or more classes with the training set containing precise labelled instances of all the classes. Weka offers two one-class algorithm: LIBSVM which is a wrapper over the libsvm library [38] and oneClassClassifier [36]. The first implements Support Vector Machine functions while the second is presented as a meta-classifier and no further information are released. We tried both the algorithms but their results were almost the same and then the next section will cover only the LIBSVM tests.

Test 01

Training: 10000 snapshots taken at random from set A. Test: 500 snapshots from A (non included in the training set) and 500 snapshots from B. Result: 500 snapshots from A were correctly marked as coming from A, 100% accuracy, 498 snapshots from B were marked as coming from A, 0.4% accuracy and a 99.6% of error.

Test 02

Same approach of test 01 but with a subset of features. We removed from each snapshot all the field strictly legated to the single moment (too influence by the environment and the movements). These fields were: the proximity, the light, the battery temperature, the pressure and the accelerometer. Results were almost the same of Test 01. The use of a subset has not led to improvements.

Test 03

Instead to use a single snapshot as the smallest training unit we performed an average over 10 consecutive snapshots and used the average as the training unit. From the original 17280 we obtained then 1728 new snapshots. Each value represent is an average over 60 seconds. Also this time the results were almost the same of Test 01 with no improvements over the accuracy. Further this approach could not be used in a single snapshot evaluation for identity reinforcement as we did in the Identify Reinforcement Module.

10.1.1 Tests results

We obtained really poor accuracy and discrimination capacity in all the tests we made. We tried different configurations but always with no improvements. Even radically changing a snapshot picket from set A, in order to vary it from normal range of values, didn't bring it to be recognised as a foreign snapshot. These recognising problems are generated by the very high variations in time that are present in the snapshots. Consecutive snapshots could have very different range of values and this doesn't permit to the algorithm to set proper thresholds. Sometime, the value of

a field changes so much to cover practically every possible value the sensor could return. The consequence is that every value returned from another user belongs howsoever to the already seen values and then is recognised as coming from A. For example, let's think about the CPU and RAM values. For both of them the possible range goes from 0 to 100 and in the snapshots almost all the range is covered. Further, despite what could seem obvious, CPU and RAM values are uncorrelated. There are snapshots where RAM occupation is high (over 70%) while CPU utilization is low (under 20%) and vice versa. This missing of correlation does aggravate the situation and makes it even more difficult the overall process. The same behaviour happens with calls done and received. There are snapshots with 0 calls and snapshots with one or more calls. To end up, no attribute or set of attributes is so user's habit representative to permit to distinguish if a snapshot was calculated by a foreign user. Maybe if we were able to elaborate data using values over a period of time the results might have been different but this possibility do not fit with our requirements and purposes. To conclude, with the WEKA library we were unable to reach a discrimination capacity comparable with our statistical approach.

10.2 Malware Discrimination

Second point we want to investigate is if it possible to train a system to understand, from the analysis of a single punctual snapshot, if a mobile device is infected by a malware. With infected system we intend the presence of an application which contain, in addition to its normal code, also a malicious hidden part. To distinguish an infected device from a clean device we need to perform some kind of comparison between the data generated by the two systems. To obtain objective data we decided to create our database from the same device but with two version of the same application installed. This differences has led to two distinct sets of data:

1. Set "Clean": on device A are installed only the manufacturer preinstalled applications and a clean, non infected, external application of our choice;
2. Set "Infected": on device A are installed only the manufacturer preinstalled applications and an infected version of the external application of our choice.

This is a two classes discrimination problem. We want to train the system with a subset of Clean and Infected and then see if the system is able to assign classify correctly a new snapshot.

We repeated the tests with three applications, all presented in the Snapshot Module: 24H Analog Widget (clean and with SMS to premium number infection), Currency Converter (clean and with contacts and other personal information sent to a remote server) and MiniStock (clean and with memory and cpu saturation infection). We collected one week of snapshots for each application: 604800 snapshots with the clean sample and the same amount with the infected sample.

Weka offers many algorithms to handle the two classes problem such as K-Means, SVM, Bayesian Networks, Decision Tree and Naive Bayes. We tested all of them with only a small difference in the results obtained. We will cover just some of them in the next part.

Test 01

Application: 24H Analog Widget infected with SMS to premium number. Algorithm: K-Means using 1 nearest neighbour for classification. Training: 200000 consecutive snapshots from Clean and 200000 consecutive snapshots from Infected. Test: 2230 new snapshots from Clean and 2040 new snapshots from Infected. Features: all. Result: 1890 snapshots from Clean were correctly marked as coming from Clean. 272 snapshots from Clean were marked as coming from Infected. 1820 snapshots from Infected were marked as from Clean. 220 snapshots from Infected were marked as Infected. The total accuracy index was equal to 0.494.

Test 02

Application: 24H Analog Widget infected with SMS to premium number. Algorithm: K-Means using 1 nearest neighbour for classification. Training: 200000 consecutive snapshots from Clean and 200000 consecutive snapshots from Infected. Test: 2230 new snapshots from Clean and 2040 new snapshots from Infected. Features: CPU, RAM, SMS_SENT Result: almost as Test 01 with total accuracy index equal to 0.491. The features subset didn't led to any improvement.

Test 03

Application: 24H Analog Widget infected with SMS to premium number. Algorithm: K-Means using 1 nearest neighbour for classification. Training: average over 8 consecutive snapshots. 25000 average snapshots from Clean and 25000 average snapshots from Infected. Test: 1160 new average snapshots from Clean and 1060 average snapshots from Infected. Features: CPU, RAM, SMS_SENT. Result: 1060 averages from Clean were marked as Clean. 100 averages from Clean were marked as Infected. 35 averages from Infected were marked as Clean and 1025 averages from Infected were marked as Infected. Total accuracy index was equal to 0.939. The classification with averages offer really improved performances.

10.2.1 Tests results

Test 01 showed which is the main problem when dealing with the malware analysis over a single snapshot. The infected application has in most of the cases, and this is clearly visible by manually analysing the data, the exact behaviour of the clean one. This is exactly the conduct that the malware wants to simulate. We don't know if and when the malicious code inside the application will be executed and then it will begins to change the application behaviour. Moreover, the phenomenon is not continuous in time: the malicious part can stop its work and restart immediately or after an unspecified number of snapshots as shown in Fig. 10.1.

During the training phase a subset of Infected is equal to the Clean subset. Without knowing the malware itself it is impossible to distinguish it from the base behaviour. Then again, it is not possible to understand during the test phase if the new received snapshot is from Clean or from the subset of Infected that coincide to Clean. In test 02 we used a different approach. We developed the malware and then we know exactly what to look for: a difference on the SMS number among the two behaviours. Then we restricted the features just to CPU, RAM and SMS. We included CPU and RAM because to send a SMS the malware start a background

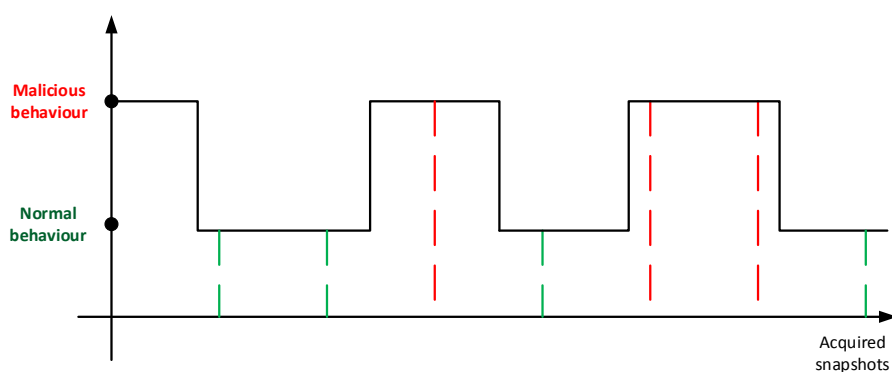


Figure 10.1: A malware can alternate its behaviour in time

hidden process and the overall performance may be effected. Again as before, the system was unable to correctly discriminate between Clean and Infected. Malware part operate too infrequently and then most of the time the SMS count is equal to 0, exactly as in the Clean set. Further some times the user send SMS and then SMS counter is increased in the Clean set and still zero in the Infected. With these deviations it is impossible to define thresholds. Test 03 is performed with averages over eight consecutive snapshots. Such averages permitted to remove, most of the time, any 0 from the SMS field in the malware snapshots. While this approach permit a very powerful discrimination over the behaviour, it is unusable for our interests for three main reasons: first it need averages over time and we want to discriminate with just a single snapshot. Second we manually removed must of the features: with a normal malware we don't know how it acts and what it does and then we cannot reduce so easily the set of features. Third, there are different kind of malware and we don't know how often they interfere with the system. If the malware works every two hours there is not improvement over the discriminatory performance if we analyse averages over two minutes intervals. Further the malware may attack on more fronts, for example it may forwards the contacts to a remote server and in the same time delete use the CPU for compute Bitcoin.

In conclusion, from our tests, both the identity reinforcement and the malwares discovery tests did not offer the desired results and the discrimination could not be computed with a reasonable accuracy. These results can be explained due the nature of the data. The punctual values contained in a single snapshot fluctuate too much in time and also a malware may act as a normal application during the learning and testing phases. Better results could be achieved working with averages and subset of features. The first is beyond what we were looking for (single snapshot classification) while the second was possible just because we made the malwares and we know their operate a priori. The use of the Weka with the available database did not lead to the desired results and the statistical approach used in ARAM has led to much better results.

Chapter 11

Conclusion

In few years Android has become the most diffused operating system among mobile devices. Due to the extremely rapid diffusion and the kind of data those devices contain - personal and financial information - the development of malwares targeting Android has grown hand in hand with the sales records. Consequently the need to protect the devices and their owners' privacy quickly captured the research community.

In this thesis we presented a complete architecture which permits to remotely monitor the device behavior and alert the owner if an anomaly is detected. Our approach, thanks to a centralized server, is able to identify if an application has a malicious behavior, especially if it contains hidden services as most of the Android malwares do.

Through the definition of specific rules it is possible to control how the device behaves in time. The user is also able to define his own policies to control its device and better meet his habits and requirements. To our knowledge we are the first to present such mechanism and our tests demonstrate how powerful can be the approach we propose.

More in detail, our system checks every package installed on the user's device to understand as soon as possible if an application can be harmful. In this way we have a very first discrimination between a normal and a tampered APK. Our tests demonstrated how it was possible to promptly discover after the installation if the application presents alterations to the APK.

The data acquired in time permit deeper uses as the ability to reinforce the login process on thirty-party websites. In the future, with the diffusion of Android wearable objects such as watches and glasses, the personal data acquired from user and environment will be greater, more accurate and therefore with a greater chance of discrimination.

The overall system effectiveness is the result of the contribution of each module. In fact, each module has a different purpose and it is able to discover a specific malware attack. While a single module can fail and not be able to readily discover a malicious behavior, the multi-control approach ensures a higher discovery capability with less chance of error.

The ARAM system has been designed as a prof of concept. Our intent was to analyse device's logs and demonstrate how these were influence by a malware presence. First analysis showed how problematic was to discriminate the behaviour

from a single snapshot. Then we radically changed the way we operated and we began to monitor the behavior over time, recording what the typical behavior of the device was. The huge mass of data collected from single device has allowed therefore to define typical behavioural profiles. Only then, we were able to compare snapshot recorded in the short term with these profiles and capture anomalies. This demonstrate how our initial idea was correct but we had to change the time intervals on which we were working on.

We tested ARAM with a relative small set of devices and a very limited infrastructure. These conditions have been imposed by the tools at our disposal. If ARAM will ever reach the market and adopted for the use in real world then its implementation, hardware and software, must be revised to serve millions of devices and scale accordingly. Then, in future works such aspects must be investigated if we want a system truly usable in the real world.

ARAM collects many personal information during its operations and many users may be frightened from this. Therefore it is important that the end user understands the benefits introduced by this relative loss of privacy otherwise the ARAM diffusion will be greatly compromised.

Bibliography

- [1] ProGuard Homepage
<http://proguard.sourceforge.net/>
- [2] BadLepricon malware caught stealth-mining bitcoin in Android apps
<http://www.theguardian.com/technology/2014/apr/25/badlepricon-malware-bitcoin-mining-android-apps>
- [3] Hacky Bird: Malware Infests Flappy Bird Alternatives
<http://www.tomsguide.com/us/malware-disguise-popular-apps-mcafee,news-19048.html>
- [4] Lok Kwong Yan and Heng Yin, "DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis". In Proceedings of the 21st USENIX conference on Security symposium (Security'12). USENIX Association, Berkeley, CA, USA, 29-29.
- [5] Y. Zhou, Z. Wang, W.Zhou, X. Jiang, "Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets". In Proc. of Network and Distributed System Security Symposium (NDSS), 2012.
- [6] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang, "RiskRanker: scalable and accurate zero-day android malware detection". In Proceedings of the 10th international conference on Mobile systems, applications, and services (MobiSys '12). ACM, New York, NY, USA, 281-294
- [7] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner, "Android permissions demystified". In Proceedings of the 18th ACM conference on Computer and communications security (CCS '11). ACM, New York, NY, USA, 627-638.
- [8] William Enck, Machigar Ongtang, and Patrick McDaniel, "On lightweight mobile phone application certification". In Proceedings of the 16th ACM conference on Computer and communications security (CCS '09). ACM, New York, NY, USA, 235-245.
- [9] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket2. NDSS '14, 23-26 February 2014, San Diego, CA, USA
- [10] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2010. TaintDroid: an information-flow

- tracking system for realtime privacy monitoring on smartphones. In Proceedings of the 9th USENIX conference on Operating systems design and implementation (OSDI'10). USENIX Association, Berkeley, CA, USA, 1-6.
- [11] ZHAO, M., ZHANG, T., GE, F., YUAN, Z.. RobotDroid: A Lightweight Malware Detection Framework On Smartphones. *Journal of Networks*, North America, 7, apr. 2012.
- [12] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for Android". In Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM '11). ACM, New York, NY, USA, 15-26.
- [13] Stephen Feldman, Dillon Stadther, and Bing Wang, "Manilyzer: Automated Android Malware Detection through Manifest Analysis" In proceeding of Networking and Systems REUNS 2014
- [14] Ryo Sato, Daiki Chiba, Shigeki Goto, "DETECTING ANDROID MALWARE BY ANALYZING MANIFEST FILES" In PROCEEDINGS OF THE ASIA-PACIFIC ADVANCED NETWORK. <http://dx.doi.org/10.7125/APAN.36.4>
- [15] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu, "DroidMat: Android Malware Detection through Manifest and API Calls Tracing". In Proceedings of the 2012 Seventh Asia Joint Conference on Information Security (ASIAJCIS '12). IEEE Computer Society, Washington, DC, USA, 62-69.
- [16] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Javier Nieves, Pablo G. Bringas, and Gonzalo Álvarez Marañón, "MAMA: MANIFEST ANALYSIS FOR MALWARE DETECTION IN ANDROID". *Cybern. Syst.* 44, 6-7 (October 2013), 469-488.
- [17] Naser Peiravian and Xingquan Zhu, "Machine Learning for Android Malware Detection Using Permission and API Calls". In Proceedings of the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence (ICTAI '13). IEEE Computer Society, Washington, DC, USA, 300-305.
- [18] Ashkan Sami, Babak Yadegari, Hossein Rahimi, Naser Peiravian, Sattar Hashemi, and Ali Hamze, "Malware detection based on mining API calls". In Proceedings of the 2010 ACM Symposium on Applied Computing (SAC '10). ACM, New York, NY, USA, 1020-1025.
- [19] Dimitris Geneiatakis, Igor Nai Fovino, Ioannis Kounelis, Paquale Stirparo, "A Permission Verification Approach for Android Mobile Applications", *Computers & Security*, Available online 7 November 2014, ISSN 0167-4048,
- [20] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. 2012. "Andromaly": a behavioral malware detection framework for android devices. *J. Intell. Inf. Syst.* 38, 1 (February 2012), 161-190. DOI=10.1007/s10844-010-0148-x

- [21] Adrienne Porter Felt, Elizabeth Ha, "Android Permissions: User Attention, Comprehension, and Behavior", Electrical Engineering and Computer Sciences - 2012
- [22] APK4Fun - Download Fun APK for Android
<http://www.apk4fun.com/>.
- [23] Android SDK Home Page
<https://developer.android.com/sdk/index.html?hl=i>
- [24] Android 4.3 APIs
<https://developer.android.com/about/versions/android-4.3.html>
- [25] Zhou, Yajin and Jiang, Xuxian, "Dissecting Android Malware: Characterization and Evolution", Proceedings of the 2012 IEEE Symposium on Security and Privacy - 2012
- [26] C4droid - C/C++ compiler & IDE
<https://play.google.com/store/apps/details?id=com.n0n3m4.droidc&hl=it>
- [27] Memfetch utility
<http://lcamtuf.coredump.cx/>
- [28] ptrace(2) - Linux man page
<http://linux.die.net/man/2/ptrace>
- [29] Android developer team. Documentation on the complete structure of Android OS
<http://developer.android.com/tools/projects/index.html>
- [30] Android developer team. Lifecycle of Android application
<http://developer.android.com/reference/android/app/Activity.html>
- [31] Android developer team. Statistical grow in the usage of Android
<http://developer.android.com/about/index.html>
- [32] Android developer team. Structure of the Architecture of Android
<http://source.android.com/devices/tech/security/>
- [33] <http://fortune.com/2013/12/13/apple-ios-android-fragmentation/>
- [34] Android gets 97% of malware
<http://fortune.com/2013/04/14/android-gets-97-of-malware-apple-ios-58-of-enterprise/>
- [35] <https://www.apachefriends.org/it/index.html>
- [36] oneClassClassifier: Performs one-class classification on a dataset
<http://weka.sourceforge.net/packageMetadata/oneClassClassifier/>
- [37] F-Droid — Free and Open Source Android App Repository
<https://f-droid.org/>

- [38] LIBSVM – A Library for Support Vector Machines
<http://www.csie.ntu.edu.tw/~cjlin/libsvm/>
- [39] AnagramSolver: Google Play Store
<https://play.google.com/store/apps/details?id=com.as.anagramsolver>
- [40] AnagramSolver: F-Droid
<https://f-droid.org/repository/browse/?fdfilter=anagram>
- [41] An open-source API for the Android Market
<https://code.google.com/p/android-market-api/>
- [42] Forecast: PCs, Ultramobiles and Mobile Phones, Worldwide, 2010-2017, 4Q13 Update
<http://www.gartner.com/newsroom/id/2645115>
- [43] IDC Study: Mobile and Social Connectiveness
<http://newsroom.fb.com/news/2013/03/idc-study-mobile-and-social-connectiveness/>
- [44] Market Share Analysis: Mobile Phones, Worldwide, 4Q13 and 2013
<https://www.gartner.com/doc/2665319>
- [45] Android-apktool: a tool for reverse engineering Android apk files
<https://code.google.com/p/android-apktool/>
- [46] AnTuTu Benchmark, a comprehensive Android Benchmarking application
<http://www.antutu.com/index.shtml>
- [47] Signapk: onboard apk signing script for android devices
<https://code.google.com/p/signapk/>
- [48] Hacking Android APKs or “how do I create my own Android trojan?”
<http://i-proving.com/2013/05/17/hacking-android-apks-or-how-do-i-create-my-own-android-trojan/>
- [49] Android Malware Injection into Original Apps
http://insiderattack.blogspot.it/2013/09/android-malware-injection-into-original_5.html
- [50] JSON (JavaScript Object Notation)
<http://json.org/>
- [51] ro0ted Injecting Custom Code Into Android APK’s
<https://www.cyberguerrilla.org/a/2013/?p=12641>
- [52] Reverse Engineering Android: Disassembling Hello World
<http://blog.apkudo.com/2012/10/16/reverse-engineering-android-disassembling-hello-world/>
- [53] An assembler/disassembler for Android’s dex format
<https://code.google.com/p/smali/>

- [54] Titanium Backup
<https://play.google.com/store/apps/details?id=com.keramidas.TitaniumBackup>
- [55] SetCPU
<https://play.google.com/store/apps/details?id=com.mhuang.overclocking>
- [56] Weka 3: Data Mining Software in Java
<http://www.cs.waikato.ac.nz/ml/weka/>
- [57] DEBIAN URGING USERS PATCH LINUX KERNEL FLAW
<http://threatpost.com/debian-urging-users-patch-linux-kernel-flaw/106516>
- [58] Android Open Source Project - Issue Tracker
<https://code.google.com/p/android/issues/list>
- [59] Yet Another Android Master Key Bug
<http://www.saurik.com/id/19>
- [60] Android Master Key Exploit – Uncovering Android Master Key That Makes 99% of Devices Vulnerable
<https://bluebox.com/technical/uncovering-android-master-key-that-makes-99-of-devices-vulnerable/>
- [61] Android Exploit Found (Bug 8219321 & 9695860) !!! Fix Available
<http://ajqi.com/android-exploit-found-bug-8219321-9695860-fix-available/>
- [62] Second Android signature attack disclosed
<http://www.h-online.com/open/news/item/Second-Android-signature-attack-disclosed-1918061.html>