Università
Ca'Foscari
Venezia

Master's Degree programme in

Computer Science

Final Thesis

Remote Debugging Detection in Android

Supervisor
Prof. Paolo Falcarin

Graduand
Mirco Venerba
Matriculation Number 872653

Academic Year
2022 / 2023

Contents

Abstract

The purpose of this master thesis is to design a software library that is able to analyze and monitor the different types of data that an Android device makes available (such as system calls and their sequences, sensors data which the most important information come from the accelerometer and gyroscope, the different settings of the device that we can set and much more) and through these, this library will be able to identify anomalies that can be indicators of security problems or indicators of the presence of a possible debugger.

To implement all of the above I had to design and build an architecture consisting of an Android library which has to be imported into the application to be traced to extract the data and a web server which receives and analyzes the data from the Android application and if it finds any suspected sequence or some anomalies, through an intermediary interface, the administrator decides which operation to take as stopping the network communication of that device because it is considered compromised or putting that device in a devices warning level list.

The Android library consists of two execution modes (the first consisting of the current application verification mechanism and the second consisting of a training phase for the web server), several interconnected modules (system call monitoring using the Ptracer tool positioned between the application and the kernel being able to intercept any interaction and monitoring of the multiple data that the Android device can provide us such as sensors, settings, type of recharge, applications that can be debugged and much more) and finally the communication module implemented via web socket using two different channels for ptracer logs and android logs.

The web server can be seen as the central node of the entire architecture and in fact has the task of receiving from the device to be analyzed all the data described in the previous paragraph. From these information, a model was created consisting of all the possible combinations that identify normal, or rather ideal, behavior of an Android application and this is used then to be able to compare each future execution. It can be observed that the information collected in the model during the training phase are

extremely useful for detecting if an external actor is trying to debug, tamper or violate the application since such attempts would alter its normal behavior, execution speed and much more.

List of figures

1. Introduction

This chapter aims to provide a general introduction of the multiple topics that will be covered within this thesis in order to conceive, design and develop the entire architecture presented in the abstract.

In recent years we are witnessing huge technological transformations characterized above all by the increasingly widespread use of devices such as smartphones, tablets and smartwatches and all of these have literally invaded the daily lives of millions of people. These, thanks to the many software applications made available to users, that can be used for various purposes including leisure activities such as video games and streaming, work activities that process more and more sensitive data up to applications that perform operations of considerable importance that involve the movement of money such as purchase operations on e-commerce platforms or bank transactions.

However, this also brings a disadvantage that consists of large quantities of sensitive information, such as personal information or credit cards but in general we can talk about network traffic, sensor data, device settings and much more, made available and collected by the applications that we use every day and this makes Android a potential target for malicious users interested in obtaining sensitive information, device control and possibly software piracy.

A main figure, in addition to the sensitive information described in the previous paragraph, is made up of system calls which represent the only mechanism capable of interfacing a user process with an operating system service, such as file system management, networking and much more. This is because a process without making system calls can only perform calculations and operate on its own memory which is volatile.

Consequently, being able to map any interaction between the application and the kernel is an effective way to have a complete view of the actual behavior of the application to be monitored because any action, considered potentially harmful, must involve at least one system call.

Consequently, the purpose of this thesis is to design a library that is able to analyze, monitor and correlate the multiple types of data that an Android device makes available, both at a low level such as system calls and at a high level as can be the sensor data and the different device settings that we can set, and through these be able to identify anomalies that can be indicators of security problems such as reverse engineering or debugging attempts.

1.1. MATE attack

A very important figure when we talk about anomaly detection and debugger detection consists of Man-At-The-End type attacks or also known as MATE attacks.

This category of attacks is particular, it consists of attacks that are difficult and complex to analyze, model, evaluate and prevent and this is due to two main reasons. The first consists in the fact that the attacker is a human being and behind this there is a combination of motivation, creation and ingenuity while the second consists in the fact that the attacker has physical, unlimited and authorized access to the device to compromise.

The key point of information security is that all major protections and security resist an attacker for a certain period of time, more or less long, but then after a period of study, motivation and money all protections will eventually fall.

For this reason, in all the digital resources widely available today, starting from the workplace to the personal one, up to home networks, the cloud and finally the Internet of Things, traditional IT security is no longer sufficient.

In fact, a first example of a very simple and nowadays widely used attack is reverse engineering, which consists of the process by which a detailed analysis of the structure and functioning of the target to be compromised is performed, whether physical, software or hardware, in order to extract its implementation and design specifications.

Since this type of cyber attack not only allows you to extract the specifics, both of implementation and design, but also vulnerabilities that can be subsequently exploited by the attacker to compromise the target, this is also strictly prohibited by the terms of use of the software and as a result, multiple types of measures are created and designed to prevent and make them more complex.

Accordingly in the literature, mainly three techniques used to prevent MATE-type attacks are described. These include obfuscation, tamper proofing and watermarking.

Obfuscation consists in transforming an input source into another output source with the same functionality but more complex to analyze and modify for the attacker. Tamper proofing consists in inserting some automatic controls within the input source code to verify its behavior to verify that it has not been compromised or altered and if this is detected, behave accordingly. Finally there is watermarking which consists in inserting other unique information encoded within the input source and these can subsequently be extracted to obtain information regarding the origin, provenance and consequently the uniqueness of actual copy.

1.2. Anomaly detection

The second fundamental topic to be discussed in order to be able to design, develop and subsequently write my thesis is anomaly detection which consists in the identification of rare events, elements or observations that are suspicious because they differ significantly from the standard behaviors or models that are created in the training phase.

In fact, the structure of my project consists of a web server that analyzes the data it receives and an Android device that extracts and forwards the multiple types of data it may have.

The latest generation Android devices have access to multiple types of data, such as sensors, settings, data flows and much more, and being super connected to the internet and other high tech devices exponentially increases the risk of disclosure of sensitive information.

For this reason, the Android operating system provides a modular permission system that ensures that every application that runs explicitly requests what permissions are required in order to function correctly. The problem with this thing is that once the authorization has been granted, we don't know what happens next, that is, once the authorization has been granted to a specific component, we have no way of knowing whether the application is using it or not this privilege when requested or if it is constantly listening and we don't even know what is done when this data is collected because they can also be sent without knowing it to a specific remote server.

For these reasons, another important figure in this area is the intrusion detection system or also better known as IDS. It is a tool designed to perform continuous security monitoring, in order to identify, in advance, all attacks on the entire network infrastructure. To be able to implement it, we can rely on multiple types of data and in relation to those chosen, we can identify different types of IDS.

The first type consists of the Host Based Intrusion Detection System, or HIDS, which consists of an analysis of the host for intrusions. Intrusions are detected by analyzing system log file structure, system calls, file system changes, and more. The second type consists of the Network Intrusion Detection System, or NIDS, which consists of an analysis of the network traffic structure to identify intrusions, allowing you to monitor not only the single host, but even an entire network by capturing network traffic and looking for traces of attacks. Finally there is the Hybrid Intrusion Detection System which consists of a system that combines the two approaches seen before, in fact information retrieved from agents running on hosts is integrated with information fetched from the local network.

This thesis focuses on the host-based intrusion detection system, HIDS, as the data that is analyzed by the web server is extracted from the Android device to be monitored.

2. Related work

This second chapter wants to describe what is the current state of the art of the two main topics that are discussed within this thesis and these are MATE attack and anomaly detection.

In recent years we have been witnessing enormous technological transformations characterized above all by the increasingly widespread use of devices such as smartphones, tablets and smartwatches and all of these have literally invaded the daily lives of millions of people. These, thanks to the numerous software applications made available to users, can be used for various purposes including leisure activities such as video games and streaming, work activities that process increasingly sensitive data up to applications that perform operations of considerable importance involving the movement of money as purchase operations on e-commerce platforms or bank transactions.

For this reason, these two topics have seen growing interest in recent years and consequently the implementation of new prevention and protection techniques are always on the agenda. In fact, within this thesis, a new approach will be described which, through a training and model creation phase, will be able to prevent any malicious attacks. Consequently, in order to design and develop this platform, we had to read up on the current state of the art of the following two topics: MATE attacks and anomaly detection.

2.1. MATE attack

A solution related to the protection of computer systems that is acquiring a fundamentally important part concerns the protection of software. Usually, when we think of a cyber attack we tend to think that the attacker is a person external to the system being attacked, assuming that the people inside that system are trusted users and in the absence of malicious intentions and instead in MATE attacks the final user is the attacker.

In this scenario just described, software protection plays a very important role, in fact an attacker could violate the intellectual property linked to a piece of code by illegally appropriating the developed algorithms, understand how the code works through the process of reverse engineering, copy and disclose without any authorization codes or parts of it in violation of copyright laws, tamper with and modify a part of the code so that it performs instructions for which it was not designed, take advantage of some vulnerabilities present in the code and much more.

For this reason, a first technique consists in obfuscation, i.e. the modification of the code or the flow of execution of the code itself while maintaining the original behavior unaltered to increase its complexity in order to make the analysis and study more complex and Tigress is the example that we will analyze below.

Tigress is an open source tool that enables source code obfuscation written in C and C++ and is available for the Unix platform. This tool provides several obfuscation methodologies among which you can observe the flattening of the control flow graph, the merging of two functions into one, the division of a function into smaller functions, the substitution of arithmetic expressions composed of numbers integers with more complex expressions, the substitution of variables defined as integers with more complex representations, the creation and use of predicates and opaque variables and anti-branch analysis.

In addition to obfuscation, there are other techniques for software protection and these are mainly device attestation with which the device properties are verified by observing that they are legitimate and not counterfeit, dynamic code renewability with which created a tool chain capable by following multiple policies of dynamically modifying the code to make the attack more complex and finally the self debugging with which other debuggers are prevented from connecting to the application as one is connected from the beginning occupying the only position available.

An example of this implementation is the ASPIRE project which consists of a software protection tool chain composed of the following functionalities: data hiding, algorithm hiding, anti tampering, remote attestation, code renewability and self debugging.

Finally there is the OWASP Mobile Application Security Testing Guide, or also known as MASTG, which is a comprehensive handbook for mobile application security testing and used as a foundational learning resource covering a variety of topics from operating system internals to advanced reverse engineering techniques, identifying the root of a device, checking executable files, checking timers and much more.

2.2. Anomaly detection

We use our devices every day, from smartphones to smartwatches and internet of things devices, as if we are always protected from everything and everyone, we install applications without checking what information they will take from our device, we leave sensors such as position sensors and various connections always enabled and we pay everywhere with a simple click from the phone.

Consequently, given the enormous expansion of these technologies, the level of confidentiality of the information that our device must extract, analyze and process has had an increase in interest as their privacy and their security are always on the agenda.

A first step, even if not sufficient nowadays, is the static analysis which consists in the study and analysis of the code of a program through an evaluation process based on the structure, form and content and therefore does not require the execution of the program you are looking at. Since this thesis is based on an Android project, any app in this operating system is programmed, compiled, installed and executed following this scheme: java file -> class file -> Java Virtual Machine -> Just In Time Compiler.

A first example of static analysis is the one that analyzes portions of code to identify the parties that acquire sensitive data and their communication to the outside. An example of software that implements static analysis is ProfileDroid which, through this type of analysis, is also able to observe the multiple authorizations that are requested from the user.

However, this first approach is not complete because the application can dynamically download and execute code from a remote server and consequently pass control over the static analysis without problems.

From this a new approach was born, i.e. dynamic analysis which consists of an overview of the various levels of abstraction of the behavior of the application to be traced in order to be able to observe the complete path of all the extracted and communicated information in real time. TaintDroid is a first example of software that allows you to perform the checks previously described and in fact by exploiting a virtualized execution environment it allows real-time analysis of this data and through this identify those that can be considered private, observe variables, files that are opened and written, methods that are executed and much more but unfortunately it has a disadvantage which consists in being able to monitor only the interpreted instructions and not the native ones.

To solve the problem of downloading and executing code from a remote server, we can consider the DroidTrace software which implements both types of analysis, the static one to observe the portions of code that are being connected externally to execute what has just described and the dynamic one to monitor the behavior of the application.

A further evolution consists in sandboxing which consists in inserting the application inside a bubble in order to be able to easily identify and detect any interaction with the outside world and Aurasium is an example of this and in fact it is positioned at the user level between the code and native libraries used. One disadvantage about sandboxing is that this level of protection is easy to identify.

A fundamental part in the analysis and collection of data in the field of anomaly detection consists in the invocations of system calls to detect and identify malicious sequences or portions of code that acquire sensitive data. We can observe some theoretical results starting from finite state, non-deterministic, stack automata, up to the most modern VtPath, Dyck and VPStatic which try to solve the problem of intrusion detection through static analysis.

Consequently, this thesis wants to create an interaction and protection mechanism between an Android application and a web server. Information from multiple levels will be collected from the device starting from low-level system calls up to multiple high-level data such as sensors, settings and much more and then the web server will use this information by providing an administrator with a control panel to decide which operations to take.

3. Background

This chapter describes the multiple background concepts needed to be able to design and develop this platform. The main fulcrum consists of some libraries that are imported into the application to be traced, with which multiple types of data are extracted, low-level ones such as system calls and high-level ones such as sensors and settings.

The application that we want to trace and verify the execution must include two libraries. The first consists of a native library called Ptracer that is positioned between the kernel and the user level and it has the main function of extracting the system calls that are executed. The second one consists of a high level Java library that uses some listeners to receive and extract multiple high level information such as the gyroscope and accelerometer sensors up to the type of recharge, settings that we can set and much more.

Consequently, considering the complexity of this project, we have to analyze what can be the crucial points and among which we can immediately observe the tracing of system calls, the way that applications run in Android, the internal communication, the structure of executable files and how external libraries are imported.

3.1. System calls tracing

Android is a Linux based mobile operating system and consequently system call tracing also coincides with that of Linux. For this reason, a process tracing interface is provided based on two main components that work as follows: the first called "tracer" which allows to observe and monitor the execution of the second one which is also called "tracee".

Also, a key point to discuss regarding system call tracing is process and thread management. A process is also known as a "heavy weight process" which can define the instance of a program and in fact when it is started it is assigned a unique number called process ID, or even PID, which uniquely identifies it in the system. A thread is also known as a "light weight process" and differ from the former in the context of execution, in fact each process is made up of one or more threads which carry out the tasks of the parent process and they too, in the creation phase, are assigned a unique number called a thread ID, or even TID or SPID, which also uniquely identifies it in the system.

Consequently, the tracing mechanism is enabled through the "ptrace" system call which allows you to specify the identifier of the process or thread to be traced and once the tracing has started, no other tracers are allowed for the same process or thread.

Within the project, in order to implement the extraction of system calls in the device to be monitored, I imported the Ptracer library which is a project that supports Linux x86_64 and AARCH64 platforms and uses the ptrace command to acquire system calls and signals and subsequently, in addition to displaying them, it is also possible to decode their parameters, the stack trace and makes possible to create NFA models.

3.2. Android application

The main point to analyze in applications in all devices and operating systems is the moment of their launch and in Android, being the operating system used by the devices that are monitored with our platform, it is characterized by a Zygote process. This process is created when the operating system is started and the references of the code and resources used by the Android framework are loaded into it, such as themes, activities and much more and then Zygote process is forked every time an application is started by inserting, in addition to the references already possessed by the parent process, also the code of the application itself which will then be executed.

The second point to discuss regarding Android applications consists of the APK file, its structure and external libraries. In fact, the application that we want to trace, will import external libraries in order to subsequently extract the related data. An APK file can be seen as a ZIP archive that contains all the files and directories that make up our application such as JAVA files, resources files and compiled files. Inside it we can observe some important folders such as the meta-inf folder which contains signature files and manifest files, the assets folder which contains arbitrary files such as text, xml, html, fonts, music and video, the res folder which contains uncompiled resources such as strings, drawables and layouts and finally the lib folder which contains the compiled code for each architecture such as armeabi, arm64-v8a, x86, x86_64 and mips and in fact each of these is its own subdirectory.

The third and final point to discuss regarding Android applications is the communication between processes. In fact, a very important figure consists in the isolation of processes or in the fact that all processes in Android have separate address space and consequently no process can directly access the memory of another one and this useful to increase security and stability. However, it often happens that one process wants to access another process because the latter offers useful services. Consequently the latter must provide an interface or mechanism that allows this interaction and this is called inter process communication or IPC. Since the standard IPC mechanisms were not flexible enough, a new IPC event-based mechanism called Binder was developed for Android. In fact, every interaction such as a simple touch, the completion of an activity or an authorization to request a certain permission are events and consequently their management and their actual states are very important.

3.3. Anomaly detection

One last very important background aspect to be able to design and develop this platform is anomaly detection. The latest generation devices provide us with large amounts of data and consequently their very accurate analysis can provide us some clues about debugger detection.

A first analysis can be done regarding the data extracted from the sensors installed in the device such as light, temperature, accelerometer, gyroscope and many others and these can provide us with a first hypothesis if the current application is executed on a real device or on an emulator. However, it may happen that some devices do not have some specific sensors or that latest generation emulators allow the variation of the values of these sensors thus leading to false positives.

A second analysis can be done looking in the settings of the Android device and in fact through them, we are able to access various information that can be indicators that the device has the debugger connected. An example would be checking developer settings or checking if the Android Debug Bridge is enabled.

Finally, a final type of analysis consists of timing checks, i.e. checking the duration of system calls. When the debugger is connected and the execution reaches a breakpoint in a certain instruction, its duration will be increased and consequently having a training model on the durations and subsequences of the system calls we can ideally be able to define an ideal behavior of an android application.

Consequently our architecture would like to be able, through models created in the training phase, to analyze at a high level all these types of information that can be extracted from an Android device in order to be able to verify and compare the current behavior of the application to be traced with a previously mapped ideal behavior and take appropriate actions when inconsistencies are found.

4. Solution

This chapter wants to illustrate the entire architecture that has been designed and developed to allow remote debugging detection in Android. In order to design and develop this complex architecture I had to divide the entire project into two main components, the first consisting of some libraries that must be imported into the application to be monitored and the second consisting of a web server which analyzes the data received and in relation to it performs the appropriate actions.

Consequently, it can be observed that the whole process is composed of the following phases:

- the first phase consists in extracting the data using the two libraries that I have implemented, the first of which for low-level information such as system calls, their sequences and subsequences and the second library for the extraction of high-level information as can be the sensors, the settings, the type of recharge and much more
- the second phase consists in the creation of statistical models by exploiting the training method of the platform and using the previous data in order to have in detail an ideal behavior of an Android application
- the third phase consists in the reception by the web server of the multiple types of data extracted from the Android device to be monitored and then subsequently these will be transformed into optimized data structures to be able to process them
- the fourth phase consists in cataloging in real time mode the Android device that has been monitored, i.e. through multiple incremental macro flags that can be enabled for the different areas where a possible debugger connection can be observed
- Finally, the fifth and final phase consists in taking the appropriate actions when it is verified that an Android device has a very high alert level and these can be disconnection from the network, continuous monitoring, blacklisting and much more

When a developer wants to include my security tool in a generic Android application, the first step to do is the following, i.e. the inclusion of the two libraries that I have developed inside it. To be able to do it correctly there are several steps to do, including making some changes in the manifest file to add any permissions and references to some resources, loading the libraries into the project directory, overriding some methods such as onPause and onResume in the main file etc.

Consequently, in the sections below I will illustrate what are the changes to be performed within your code and I will also illustrate what are the types of data that will be extracted and how they will be processed by dividing everything into two main sections, the first dealing with the Android library for high-level information and the second that deals with the Ptracer library for low-level information.

4.1. Manifest file

The manifest file is the main file of an Android project and is located in the root of the project directory. This XML file has the task of describing the most important information for our project such as the package name, the minimum SDK version, the application name, the details of the different activities, the different services, the broadcast receivers, the different permissions required to work properly and much more. Below we try to illustrate the components of the manifest file contained within our project.

First let's analyze the application element tag, i.e. all those attributes that are defined within the application tag:

- android:extractNativeLibs="true": this attribute allows the package installer to extract the native libraries contained within the project's resources directory within the file system for later uses
- android:largeHeap="true": this attribute helps the application processes to be created with a larger heap as a lot of data is extracted and stored waiting for their communication with the server
- android:debuggable="true": this attribute is used to define whether the Android application can be connected to a debugger and in fact applying it to

true allows an attacker to debug the application thus facilitating access to parts of the application that must be maintained secure.

Secondly, let's analyze which receivers are used within the application. These represent the mechanism by which applications are allowed to receive intents transmitted by the system or by other applications even when other components of the current application are not running:

- recharge.RechargeReceiver: this broadcast receiver has been developed to allow the identification of the charging status of the device, in fact it allows you to check which type of charging, AC and USB, is activated or disabled

Thirdly and lastly, let's analyze what permissions are required for our application to run correctly:

- android.permission.QUERY_ALL_PACKAGES: this permission is necessary in order to extract the list of all the applications that have been installed on the device
- android.permission.INTERNET: this permission is necessary in order to implement the communication of multiple types of data with the web server via websockets
- android.permission.ACCESS_NETWORK_STATE: this permission is necessary in order to be able to extract information regarding the internet networks available at the moment and the relative connections
- android.permission.WRITE_EXTERNAL_STORAGE: this permission is necessary to be able to write information and data in files contained in the external storage waiting to be sent to the web server

4.2. Android library

The first library that I will illustrate in this chapter consists of the Android library that must be integrated within the project resources. This library takes care of extracting high-level information from the Android device to be monitored, such as data from sensors, the state of the application in relation to the application lifecycle, any applications that have the debuggable flag set to true, any types of debuggers that can be connected while the user is using the device and finally the type of recharge of the device.

4.2.1. Application lifecycle

The first type of useful information to analyze in order to identify a possible debugger connected to an application installed on an Android device consists in identifying the current state of the application to be traced in relation to the entire life cycle. In short, we would like to be able to extract the periods of time in which the application is running and the periods of time in which the application is in the background or not running.

In order to do all, it is necessary to analyze the life cycle of an Android application and this is illustrated in the figure below.

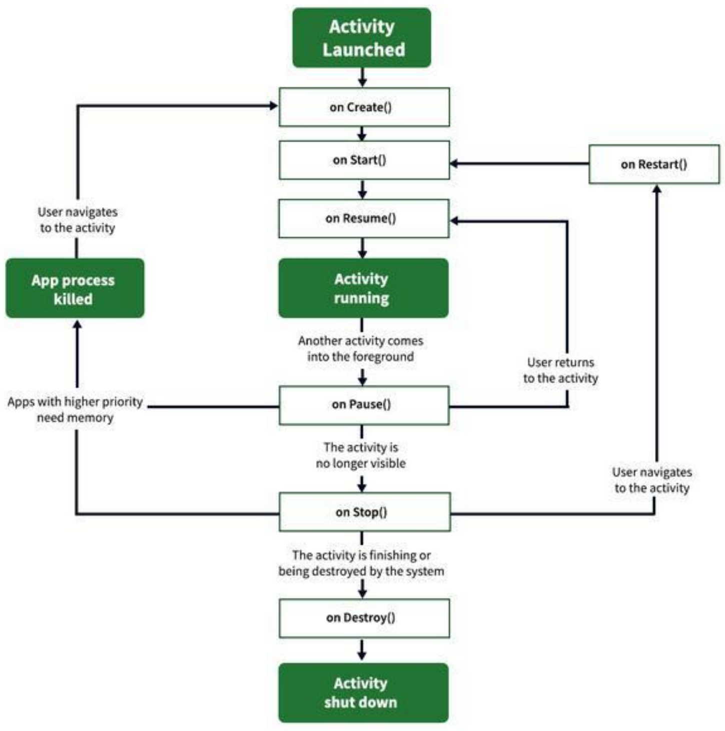
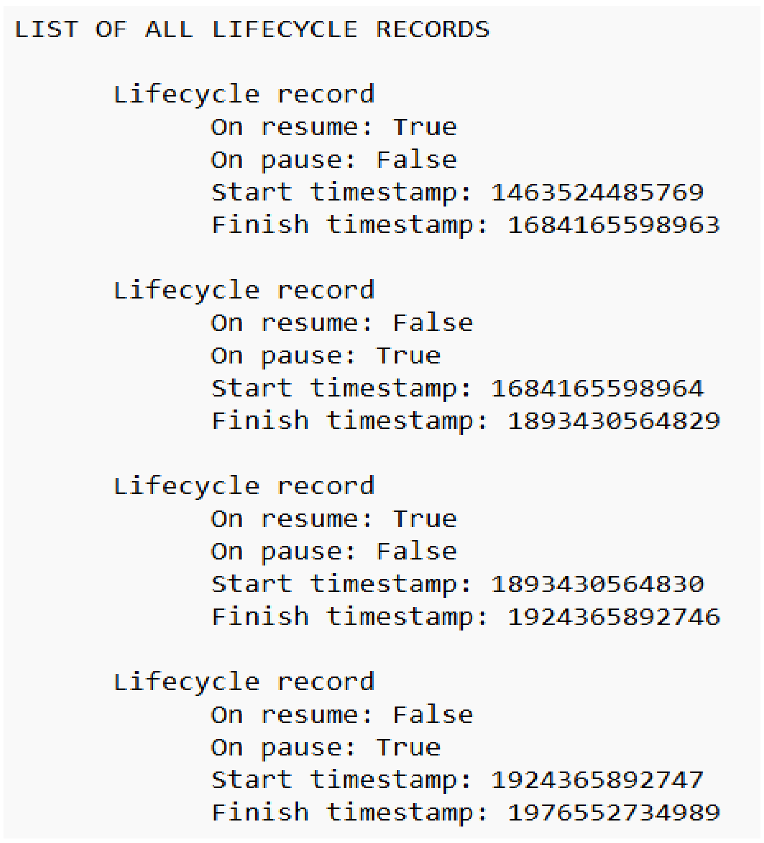

Activity Launched
on Create()
on Start()
on Restart()
User navigates to the activity
on Resume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
on Pause()
The activity is no longer visible
User navigates to the activity
on Stop()
The activity is finishing or being destroyed by the system
on Destroy()
Activity shut down

Image 1: Android application lifecycle

Two main methods that can be identified are onResume and onPause methods. The first method occurs when the activity becomes visible and can start interacting with the user while the second method occurs when the activity is paused and this can precede putting it in the background or terminating its execution.

Consequently, in relation to the data that is extracted from the Android device, an optimized data structure is created to be able to keep track of it and an example can be observed below.



```
LIST OF ALL LIFECYCLE RECORDS

      Lifecycle record
            On resume: True
            On pause: False
            Start timestamp: 1463524485769
            Finish timestamp: 1684165598963

      Lifecycle record
            On resume: False
            On pause: True
            Start timestamp: 1684165598964
            Finish timestamp: 1893430564829

      Lifecycle record
            On resume: True
            On pause: False
            Start timestamp: 1893430564830
            Finish timestamp: 1924365892746

      Lifecycle record
            On resume: False
            On pause: True
            Start timestamp: 1924365892747
            Finish timestamp: 1976552734989
```

Image 2: Lifecycle logs

These data offer us a lot of information in relation to the execution state of the application to be monitored, in fact we can observe that in the timestamp 1463524485769 the user starts the application as the onCreate method is called which in turn calls the onStart method which in turn calls onResume and it saves the timestamp. Then from timestamp 1463524485769 to timestamp 1684165598963 the application remains running and then passes in the background from timestamp 1684165598964 to timestamp 1893430564829 to then return to execution and finally it stops the execution at timestamp 1976552734989.

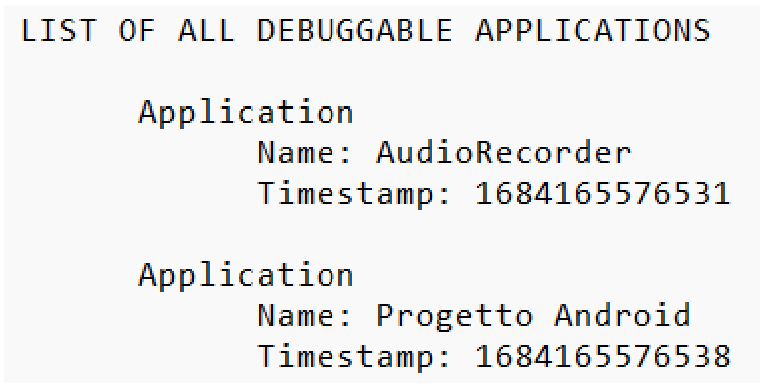
4.2.2. Debuggable applications

A second type of very important information regarding debugging detection consists in extracting the list of applications that are installed on the device by the user and checking which of these can be connected to a possible debugger. This is allowed through a flag contained in the manifest file of each application and which is called android:debuggable which makes sure that the application itself is debuggable or not.

In order to implement all this, we had to add some permissions within the Android library through the use of the manifest file and in fact android.permission.QUERY_ALL_PACKAGES helps us by allowing us, through a package manager, to extract the list of all applications installed on the device. Then, through additional flags such as ApplicationInfo.FLAG_SYSTEM and ApplicationInfo.FLAG_UPDATED_SYSTEM_APP we can understand what the type of each application is, i.e. if it is a system application or if it consists of an application installed by the end user.

Finally, as a last step, receiving as output from the previous point the list of all the applications installed on the device by the end user, we have to check which of these can be connected to a possible debugger when they are used and for this we must filter them using the ApplicationInfo.FLAG_DEBUGGABLE flag which allows to check the value of the android:debuggable flag contained within the manifest file of each application.

Below, we can see an example log of applications that are installed on the Android device with the debuggable flag enabled.



```
LIST OF ALL DEBUGGABLE APPLICATIONS

      Application
            Name: AudioRecorder
            Timestamp: 1684165576531

      Application
            Name: Progetto Android
            Timestamp: 1684165576538
```

Image 3: Debuggable applications

From these logs, we can see that two applications are installed in the Android device with the debuggable flag active and they are called “AudioRecorder” and “Progetto Android”. Consequently, only when one of these two applications is running, it can be verified that there are debuggers connected because in other cases its connection is prohibited by the manifest file.

4.2.3. Types of debugger

Another type of information that is very useful for the debugger detection problem within the application that the user is running, consists in checking and verifying the possible types of debuggers in the Android environment. The possible debuggers are of two types, the first low level and consists of the GDB debugger while the second high level one consists of the JDWP debugger.

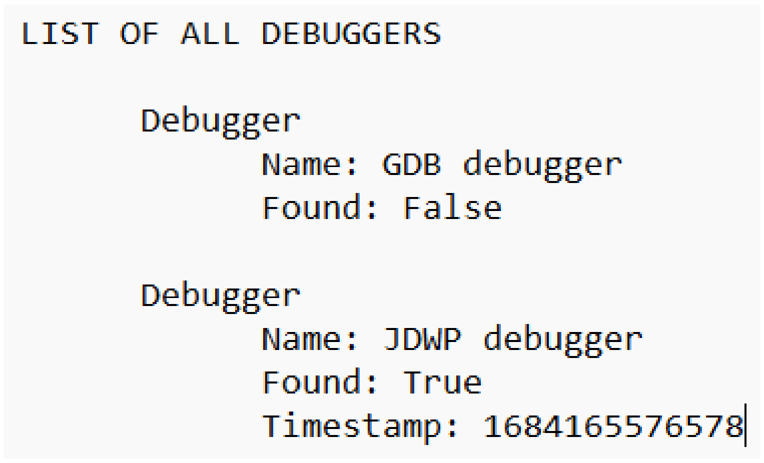
Now let's start looking at the implementation and operation of the controls for the GDB debugger identification. As written at the beginning of this chapter, the application we want to monitor must include the two libraries that we have developed, the first is low-level and the second one is high-level. The main feature of the low-level one consists in the fact that it traces the execution of the application by connecting to its PID and consequently, knowing that in the Linux environment a process can only be traced by another process, we verify the name of the connected process for verify that it is the correct process and not the GDB debugger.

In order to do all this, the proc/pid/status file comes in handy which provides us with all the information of the process with that particular pid such as the name, the status, the connected process and much more. Consequently, a very important line is the one that begins with TracerPid which indicates the pid of the process connected with the current one. If this TracerPid is equal to zero it means that there is no connected process while if it is different, this indicates the pid of the connected process.

Consequently the last step is to verify that the TracerPid is non-zero and if it is, access the proc/TracerPid/status file and extract its name. If this equals ptracer, which is the name of the low-level library, we know there can be no GDB debugger attached whereas if the name is different we know there is a possible debugger attached.

Now let's start looking at the implementation and operation of the controls for the JDWP debugger. These checks are much simpler than the previous type of debugger as the android.os.Debug.isDebuggerConnected function helps us, which directly returns a boolean value that indicates whether the JDWP debugger is connected or not.

Below we can see an example of log on the control and identification of possible GDB or JDWP debuggers connected to the application that the user is running.



```
LIST OF ALL DEBUGGERS

        Debugger
                Name: GDB debugger
                Found: False

        Debugger
                Name: JDWP debugger
                Found: True
                Timestamp: 1684165576578
```

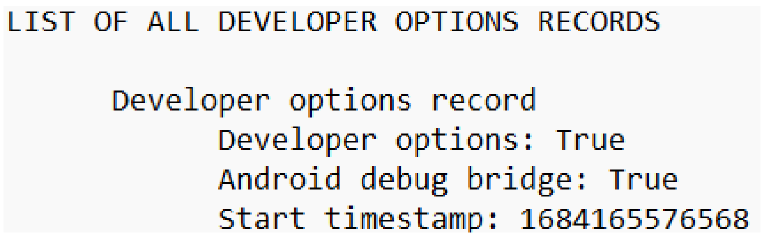
Image 4: Debuggers detection

It can be seen from the previous image that no GDB type debugger has been identified while a JDWP debugger connected to the application has been identified in the timestamp 1684165576578.

4.2.4. Developer options

Another type of information that is very useful for the debugger detection problem consists of the developer settings where a very important figure is the usb debug setting with which by connecting to a generic computer and using the android debug bridge it is possible to access and modify device information, monitor and record screen content and finally install applications which is a key factor to monitor for the entire device security.

The following image shows an example of developer settings log where we can see if the monitored device has the developer settings and the android debug bridge enabled.



```
LIST OF ALL DEVELOPER OPTIONS RECORDS

    Developer options record
        Developer options: True
        Android debug bridge: True
        Start timestamp: 1684165576568
```

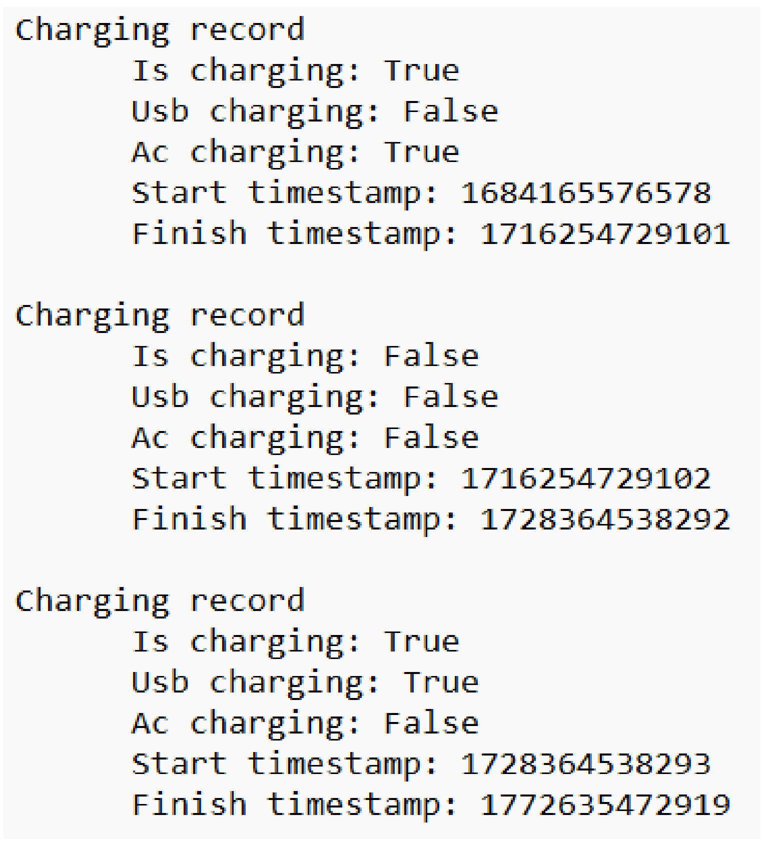
Image 5: Developer options detection

4.2.5. Types of recharge

Another type of useful information to extract for the debugger detection problem consists in verifying the active type of recharge in the device while the user is running the application to be monitored. In fact, a very important condition to verify is the presence of a possible active connection between the device that the user is using and a computer under control by an attacker. To be able to do this, the type of USB charging helps us, which indicates that a connection has been established between the device and a generic computer.

In order to verify this mechanism I had to implement a listener - receiver mechanism divided as follows: in the first part the listener is managed by activating and deactivating it in relation to the execution state of the application and consequently whether it is running or whether it is in the background or not running while the second part consists of a BroadcastReceiver which is included in the manifest file and which via the onReceive event, called whenever the event changes, saves the new state of charge of the device.

Below we can see an example log of the different types of recharge that are extracted from the device.



```
Charging record
      Is charging: True
      Usb charging: False
      Ac charging: True
      Start timestamp: 1684165576578
      Finish timestamp: 1716254729101

Charging record
      Is charging: False
      Usb charging: False
      Ac charging: False
      Start timestamp: 1716254729102
      Finish timestamp: 1728364538292

Charging record
      Is charging: True
      Usb charging: True
      Ac charging: False
      Start timestamp: 1728364538293
      Finish timestamp: 1772635472919
```

Image 6: Recharge types

We can immediately observe different periods of time in relation to the type of recharge of the Android device. From timestamp 1684165576578 to timestamp 1712254729101 the device is charging and the type of recharge is AC, from timestamp 1712254729102 to timestamp 1728364538292 the device is not charging and finally from timestamp 1728364538293 to timestamp 1772635472919 a connection is established between the device and a generic computer because the type of recharge is USB and consequently a possible debugger has been connected in this period of time.

4.2.6. Types of sensor

A final type of high-level information that is very useful for the debugger detection problem consists in the management and extraction of data from the multiple sensors that are installed in the device to be monitored, such as the accelerometer, the gyroscope, the temperature, the position, the ambient light and much more.

In relation to the debugger detection problem, a key check to verify consists in the identification of a possible connection between the device that the user is using and a generic computer under the control of the attacker. Consequently, if we assume that the attacker is debugging the application we want to monitor, it means that the attacker himself is actively using the generic computer causing the device to remain in a stationary position over time.

In order to implement all this, two main sensors must be used, the first is the accelerometer which provides us with useful information regarding the acceleration on the three axes of the device while the second sensor consists of the gyroscope which provides us with useful information regarding the inclination of the device. Consequently, in order to obtain an accurate analysis based on these two sensors, an advanced sensor called game rotation vector comes to our aid which immediately puts these data together by directly providing us with three output data: azimuth, pitch and roll which represent a measure in degrees in relation to the three axes of the device. This can be seen in the figure below.

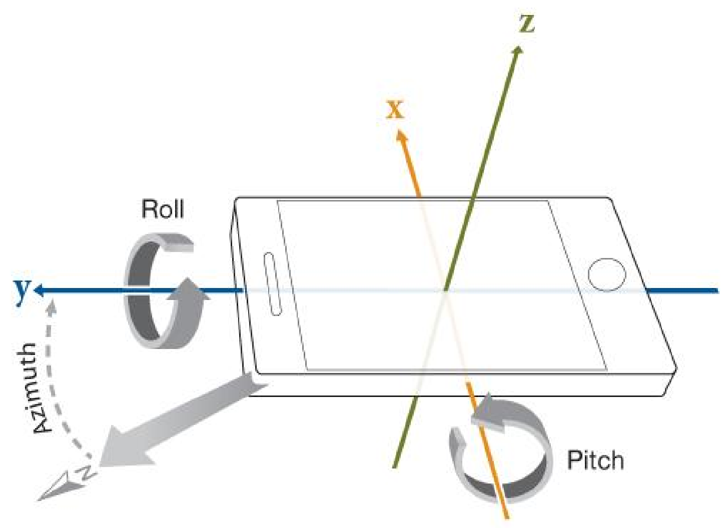
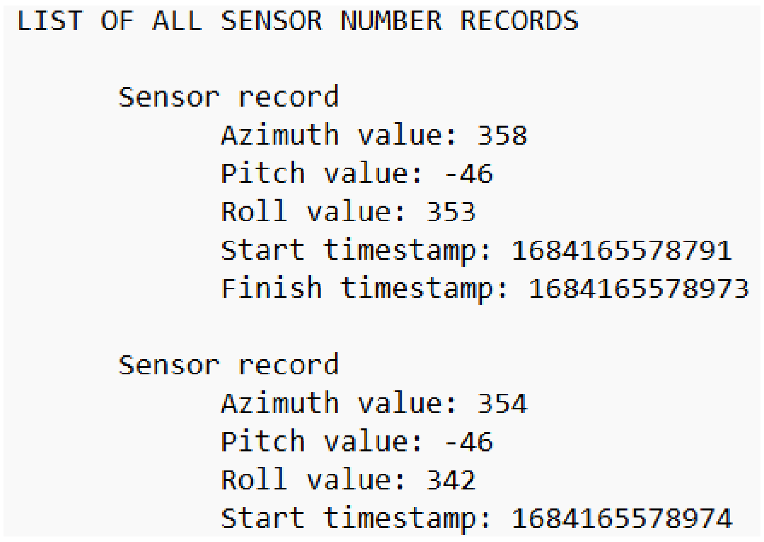


Image 7: Game rotation vector sensor

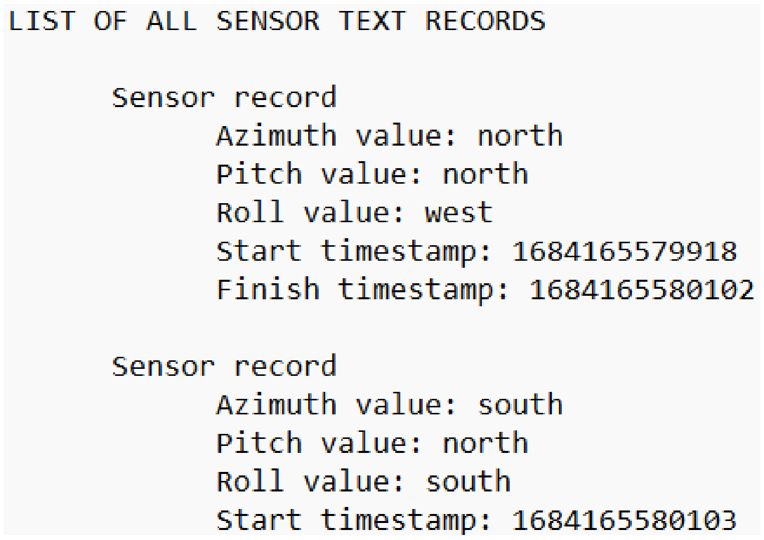
A fundamental point for the correct use and extraction of data from this sensor consists in the calibration, a process that is started a couple of seconds after starting the application, the time in which the user brings the device to a correct position and stable for the future use of the application, and it is used to be able to subsequently identify invalid device positions such as the phone too inclined, the screen not visible to the user or the phone upside down. Below we can see an example of a log of azimuth, pitch and roll data extracted from the game rotation vector sensor.



```
LIST OF ALL SENSOR NUMBER RECORDS

      Sensor record
            Azimuth value: 358
            Pitch value: -46
            Roll value: 353
            Start timestamp: 1684165578791
            Finish timestamp: 1684165578973

      Sensor record
            Azimuth value: 354
            Pitch value: -46
            Roll value: 342
            Start timestamp: 1684165578974
```

Image 8: Numeric values of sensor

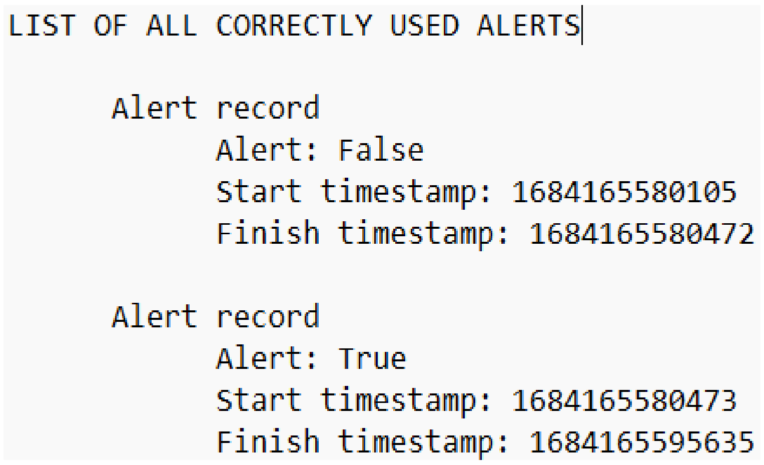
In order to observe that the phone is stationary over time, I based myself on the transformation of this data from numbers to textual values such as the north, south, east and west coordinates for each axis of the device. This is because even if the phone remains stationary over time, a slight variation of the number sensors can be observed. Below we can see an example of text logs.



```
LIST OF ALL SENSOR TEXT RECORDS

      Sensor record
            Azimuth value: north
            Pitch value: north
            Roll value: west
            Start timestamp: 1684165579918
            Finish timestamp: 1684165580102

      Sensor record
            Azimuth value: south
            Pitch value: north
            Roll value: south
            Start timestamp: 1684165580103
```

Image 9: Textual values of sensor

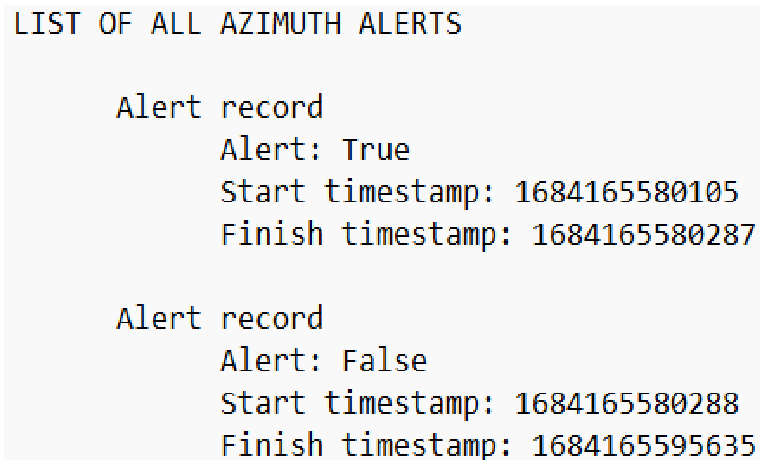
A final type of log that can be extracted from this sensor consists of the list of time periods in which the device assumes invalid positions and as explained before, these can be the phone too inclined, the screen not visible to the user or the overturned phone. Consequently these logs below are very useful to understand, when the device is positioned incorrectly, which are the axes of the device that assume an invalid position.



```
LIST OF ALL CORRECTLY USED ALERTS

      Alert record
            Alert: False
            Start timestamp: 1684165580105
            Finish timestamp: 1684165580472

      Alert record
            Alert: True
            Start timestamp: 1684165580473
            Finish timestamp: 1684165595635
```

Image 10: Correctly used sensor values

From the previous image it can be seen that from timestamp 1684165580105 to timestamp 1684165580472 the device is not positioned correctly while it is in the following period. If we check the specific logs for each axis of the device, in this case we analyze those relating to the azimuth, we realize that in the period in which the device is not positioned correctly, there is an active alert for an invalid position compared to the initial calibration.



```
LIST OF ALL AZIMUTH ALERTS

      Alert record
            Alert: True
            Start timestamp: 1684165580105
            Finish timestamp: 1684165580287

      Alert record
            Alert: False
            Start timestamp: 1684165580288
            Finish timestamp: 1684165595635
```

Image 11: Azimuth alerts

4.3. Ptracer library

This chapter aims to illustrate the second library that must be imported into the application to be monitored. This library consists of the low-level one, i.e. the extraction of system calls that occur within the device to be monitored. In fact, this library consists of two main points, the first point which consists in easily interacting with the Linux process tracking interface ptrace to acquire system call invocations and the second point which consists in having a higher level vision of application behavior by observing system call sequences and related stack traces.

In fact we can observe that everything described above is allowed by the positioning of the library, in fact it is positioned at the level immediately above the Linux kernel and thus identifies any interaction that takes place from and towards the higher levels and this can be seen in the figure below.

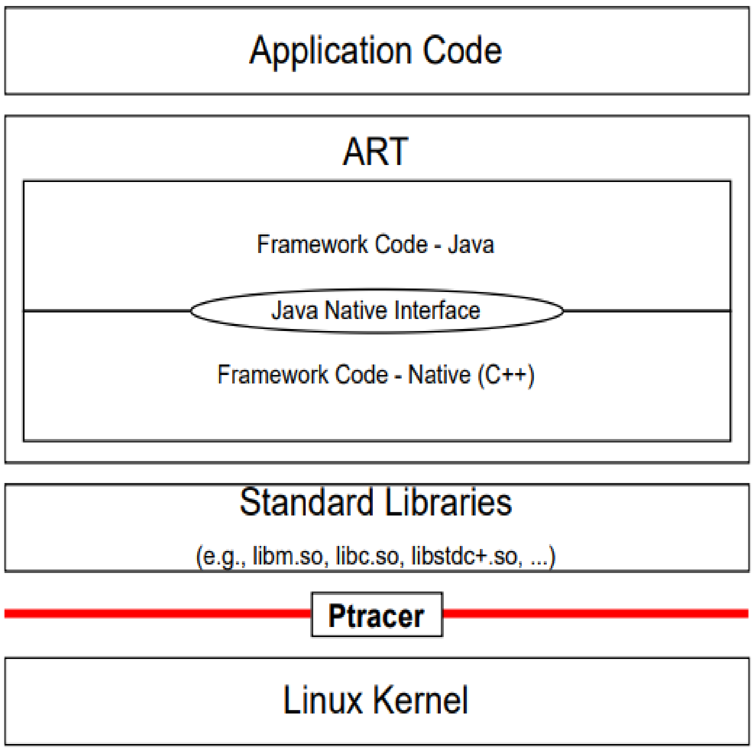
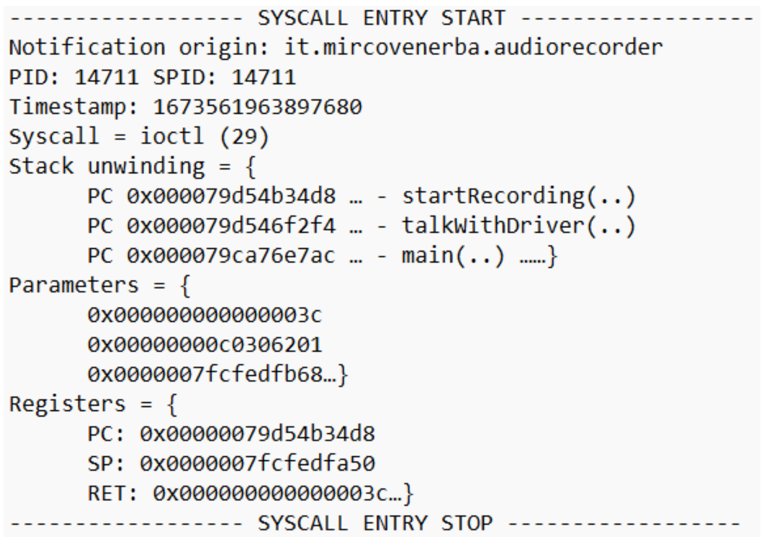


Image 12: Ptracer position

4.3.1. System call monitoring

The main function of this library, being low-level, is to identify the events belonging to system calls. In fact, when we talk about system call analysis and monitoring, we can think of many factors such as the event that triggered this call, the pid-spid combination that identifies the thread and the process that made this call, any execution start and end timestamp in order to have a more detailed overview of the relative duration, the entire stack trace that led to this call, what are the parameters that are passed to the current call, what are the values of the program counter registers, stack pointer and return pointer and finally any return value to be able to observe whether the call was successful or not. Below we can see two examples of logs where each highlights a type of event regarding system calls.

The first image represents the event when a system call is initiated and has a standard format where the event that triggered it is defined, the pid-spid identification in order to subsequently trace the relative sequences and execution paths for each pair process - thread, the stack trace that led to this system call, the syscall parameters, the value of the registers and finally the start timestamp to then be able to calculate its duration.



```
------------------ SYSCALL ENTRY START ------------------
Notification origin: it.mircovenerba.audiorecorder
PID: 14711 SPID: 14711
Timestamp: 1673561963897680
Syscall = ioctl (29)
Stack unwinding = {
      PC 0x000079d54b34d8 … - startRecording(..)
      PC 0x000079d546f2f4 … - talkWithDriver(..)
      PC 0x000079ca76e7ac … - main(..) ……}
Parameters = {
      0x000000000000003c
      0x00000000c0306201
      0x0000007fcfedfb68…}
Registers = {
      PC: 0x00000079d54b34d8
      SP: 0x0000007fcfedfa50
      RET: 0x000000000000003c…}
------------------ SYSCALL ENTRY STOP ------------------
```

Image 13: Syscall started

However for the purpose of our project, and consequently for the analysis of the durations of the system calls and for the monitoring of the execution paths, many data that are described above are not necessary, perhaps they will be in the next future evolutions of the tool. Consequently, by means of the additional flag –backtrace=false used to start this library, it is possible to reduce the size of the logs by extracting only those that are really necessary, such as the pid-spid pair, the start timestamp and finally the name of the system call. Below we can see an example of a reduced log.

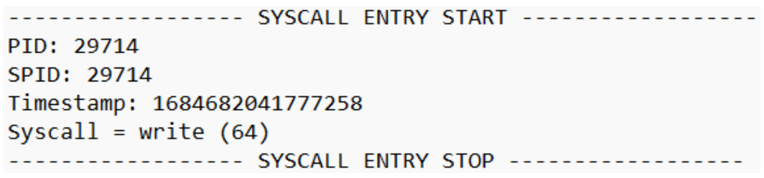

```
------------------ SYSCALL ENTRY START ------------------
PID: 29714
SPID: 29714
Timestamp: 1684682041777258
Syscall = write (64)
------------------ SYSCALL ENTRY STOP ------------------
```

Image 14: Reduced log for Ptracer

Finally, the last event to be captured concerning system calls consists in the moment in which they terminate, i.e. it is necessary to be able to capture very important information such as the end-of-execution timestamp in order to then be able to calculate the duration, the return value in order to be able to observe whether the call was successful or not and finally the pid-spid pair in order to then be able to create a logical sequence useful both in the sequence control phase and in the control for incremental executions. An example of this can be seen in the figure below.

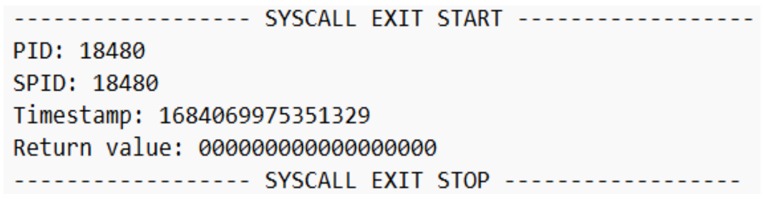

```
------------------ SYSCALL EXIT START ------------------
PID: 18480
SPID: 18480
Timestamp: 1684069975351329
Return value: 0000000000000000000
------------------ SYSCALL EXIT STOP ------------------
```

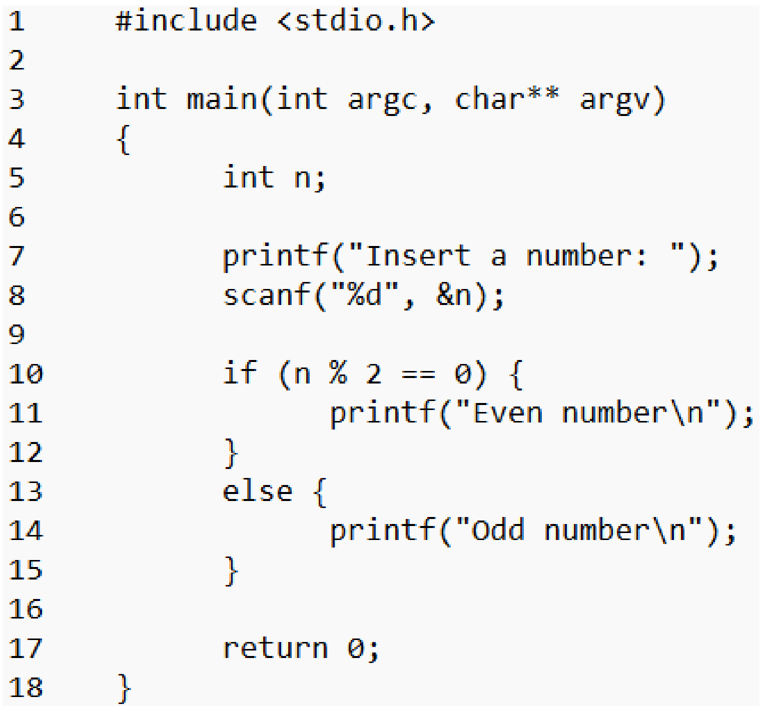
Image 15: Syscall finished

4.3.2. NFA model

Another type of information that can be extracted using this low-level library consists in the creation of finite state automata to analyze the entire execution paths that can be executed within each application through an enforcement phase where, however, they are automatically excluded any paths that lead to an error. However this functionality has not been imported into the work of this thesis as the main focus was the real time processing of the data extracted from the device in order to be able, only from an analysis of these, to make a prediction of any pattern of attack.

A fundamental point to discuss consists in choosing the nfa instead of the dfa and this happens because system calls that generate children, such as the fork instruction, will result in two ε-transitions since from now on there will be two machines executing computational activities at the same time.

A very simple example of nfa model is the one below which represents a simple program used to tell if an input number is even or odd, in fact there are only two possible execution paths and consequently only two executions are needed for the enforcement phase.



```c
#include <stdio.h>

int main(int argc, char** argv)
{
	int n;

	printf("Insert a number: ");
	scanf("%d", &n);

	if (n % 2 == 0) {
		printf("Even number\n");
	}
	else {
		printf("Odd number\n");
	}

	return 0;
}
```

Image 16: Even or odd number algorithm

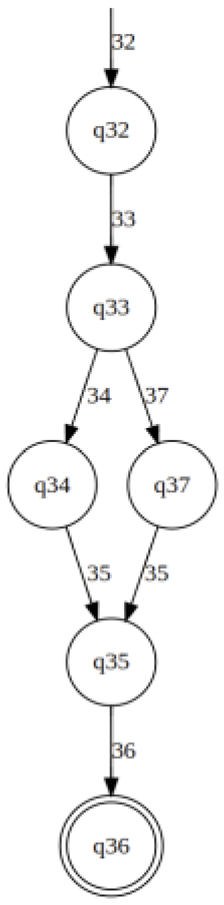


Image 17: NFA for even or odd number algorithm

If, on the other hand, we analyze more complex programs that include fork instructions, process management or a large number of activities or functions, we can observe, again below, a small piece of example of an nfa model created in relation to the audio recorder application, that we will see in the next section of the validation and testing part, which will immediately become more complex.

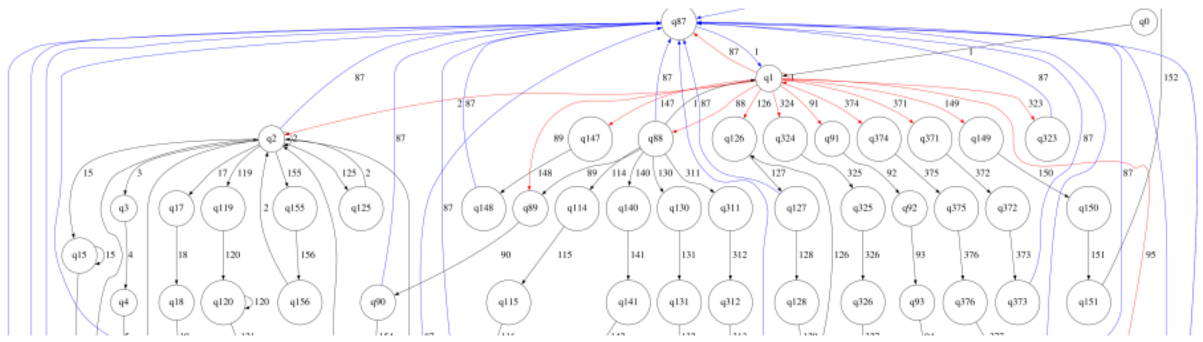


Image 18: Nfa for audio record application

4.4. Web server

Finally, the third and final component of the architecture that I designed and developed consists in the web server which has two main tasks. The first task consists in receiving the multiple types of data from the two previously described libraries and which have been imported into the application to be monitored while the second task consists in the subsequent analysis of the multiple received data.

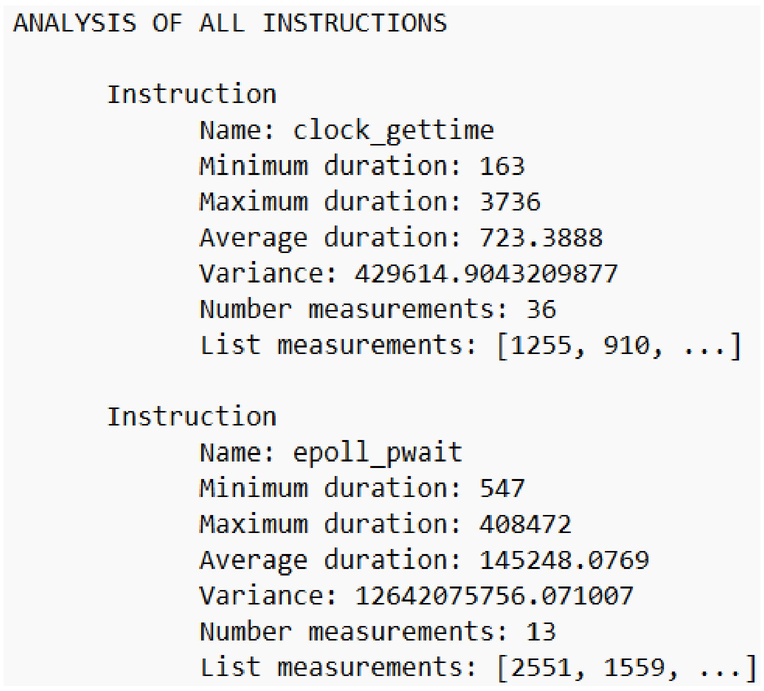
Consequently, the final task of the web server consists in the creation of advanced and complete data structures through the joint analysis of all possible types of data received from the device to be monitored, thus allowing to reach the final task of this thesis which is to be able to create an incremental security state for each connected device.

4.4.1. Instruction analysis

The first type of data that is analyzed and optimized by the web server is the analysis of each type of system calls. This mechanism consists in the reception by the web server of the events that are executed when a system call starts execution and when execution ends. A fundamental point concerns the execution flags to execute the native library as it is called by adding this additional flag "--backtrace false" which is used to not extract the entire stack trace but only the following types of data in order to minimize the amount of data exchanged between the web server and the device to be monitored: pid, spid, system call, start timestamp, end timestamp and finally the returned value.

A key point to analyze in the context of system calls consists in their duration. In fact a possible debugger connected to the application to be monitored or any breakpoints set by an attacker to study in a better way the application behavior would slow down its execution time and consequently the differences of time between the start timestamp and the finish timestamp of that system calls are greater than an average value calculated in the training phase to build a reliable model and as close as possible to an ideal behavior of a generic Android application.

That's why I created a data structure to contain the analyzes of each possible type of system call by memorizing minimum duration, maximum duration, average duration, variance and list of measurements in order to then subsequently be able to compare the current value with the average value, maximum value and ideal value calculated during the training phase. Below we can see an example of a log.



```
ANALYSIS OF ALL INSTRUCTIONS

      Instruction
            Name: clock_gettime
            Minimum duration: 163
            Maximum duration: 3736
            Average duration: 723.3888
            Variance: 429614.9043209877
            Number measurements: 36
            List measurements: [1255, 910, ...]

      Instruction
            Name: epoll_pwait
            Minimum duration: 547
            Maximum duration: 408472
            Average duration: 145248.0769
            Variance: 12642075756.071007
            Number measurements: 13
            List measurements: [2551, 1559, ...]
```

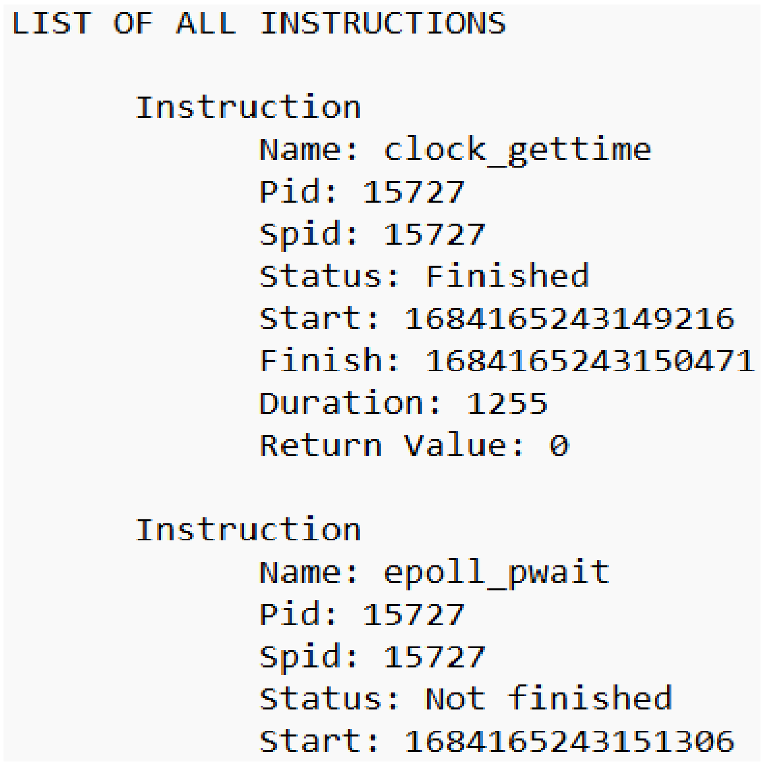
Image 19: Analysis of each instruction

From the previous image we can see two examples of system call analysis, the first consists of clock_gettime instruction and the second one in the epoll_pwait instruction where for each of them we can see minimum value, maximum value, average value, variance, number of measurements and list of measurements.

4.4.2. Sequences analysis

The second type of useful information to analyze regarding system calls consists in the study and analysis of their sequences or in the identification of possible instructions that follow the current instruction. This is very important for tracing and mapping the ideal behavior of a generic Android application in order to be able to quickly and precisely identify when there is suspicious behavior which in a nutshell translates into a sequence of calls not captured in the training phase.

A first example of a system call log file is the following, i.e. a list of the instructions that are executed with the relative details such as the start execution timestamp, the pid and spid pair, the name of the system call, any finish execution timestamp, any duration and any returned value. In addition to this list which contains all the unfiltered system calls, the same log file also contains two other filtered lists, the first which contains the system calls which have terminated and the second which contains the list of system calls which are not finished yet.



```
LIST OF ALL INSTRUCTIONS

      Instruction
            Name: clock_gettime
            Pid: 15727
            Spid: 15727
            Status: Finished
            Start: 1684165243149216
            Finish: 1684165243150471
            Duration: 1255
            Return Value: 0

      Instruction
            Name: epoll_pwait
            Pid: 15727
            Spid: 15727
            Status: Not finished
            Start: 1684165243151306
```

Image 20: List of executed instructions

A second very important example of log for mapping sequences of system calls consists of the one represented by the following image where we can see which instructions can follow the current call. If we consider that the current system call is clock_gettime, the next one can only be clock_gettime, epoll_pwait, futex, gettimeofday or read because otherwise an invalid sequence would be identified for the data obtained during the training phase which would consequently represents a suspicious behavior.

```
LIST OF ALL SEQUENCES FOR EACH INSTRUCTION

      Instruction
            Name: clock_gettime
            Next: [clock_gettime, epoll_pwait, futex, gettimeofday, read]

      Instruction
            Name: epoll_pwait
            Next: [clock_gettime, futex, recvfrom]
```

Image 21: List of possible next instructions

Finally, the last type of log that can be extracted from the application we are monitoring is the following, that is to store what the internal execution path was for each pid-spid pair. This is a very useful type of information in the long term in fact it does not consist only in informing us what the next instruction is but it provides us with information regarding what the entire sequence of instructions was for each pid-spid pair and this can be very useful in identifying not only entire paths considered safe from the data collected during the training phase but above all in identifying any similarities with paths already seen during last execution phases and in fact a subsequent analysis treated in the next section consists in calculating the similarity with respect to previous executions because they can be a symptom of an attacker who is studying the whole behavior of our application.

```
LIST OF ALL EXECUTION PATHS

      Execution path
      (15727, 15727) -> ['clock_gettime', 'epoll_pwait', 'recvfrom', ...]

      Execution path
      (16184, 16184) -> ['getuid', 'epoll_pwait', 'clock_gettime', ...]
```

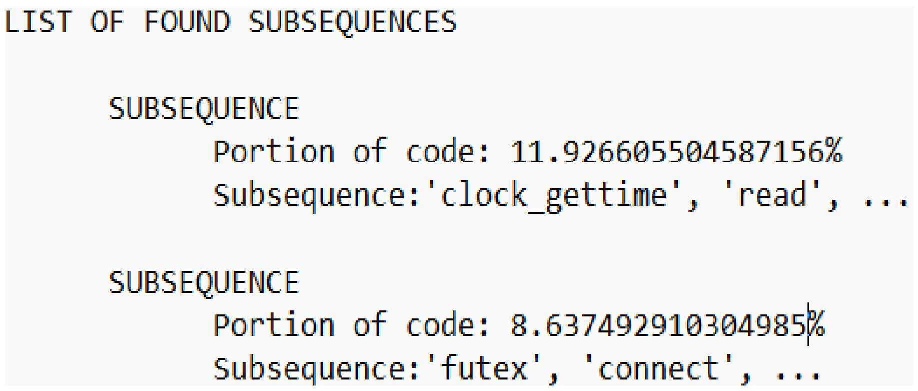
Image 22: List of executable paths

4.4.3. Incremental execution

A final type of very useful information that can be extracted from system calls consists of incremental executions or in the verification of similarity in relation to the last execution of this application. This is very useful as when there are very high similarities to the previous execution, it can mean that a possible attacker is studying the application behavior by trying and testing multiple execution paths.

An example of incremental execution could be the following, i.e. the attacker tests the behavior of the application up to a point a and then terminates it and subsequently resumes it up to the same point a but then proceeds up to a point b in the future and so on until that the whole execution is not clear. Consequently by storing the entire execution paths for each process and thread, a comparison mechanism with the last execution is created and a future development could be not only the comparison with the last execution path but the comparison in relation to all previous execution paths.

Consequently, an example of incremental execution log can be seen in the following image where we immediately see that two different subsequences have been found with the relative percentage of similarity to the entire previous execution path. In relation to the example of the following image we can conclude by confirming a total of 20% similarity and this value could be compared with a default value in the settings file to understand if it is a suspicious execution path or a safe execution path.



```
LIST OF FOUND SUBSEQUENCES

    SUBSEQUENCE
        Portion of code: 11.926605504587156%
        Subsequence:'clock_gettime', 'read', ...

    SUBSEQUENCE
        Portion of code: 8.637492910304985%
        Subsequence:'futex', 'connect', ...
```

Image 23: Incremental execution paths

4.5. Device analysis

The final task of this thesis was to identify the security status for each device that is monitored and this is allowed by analyzing the multiple types of data that are extracted from the device under consideration and then subsequently analyzed by the web server. A key point to be able to design and develop this mechanism consists in the identification of the devices which in this version, being a beta, is carried out only using the IP address which will then evolve in the next versions in a hashcode calculated in relation to the technical characteristics / identifiers of the device itself.

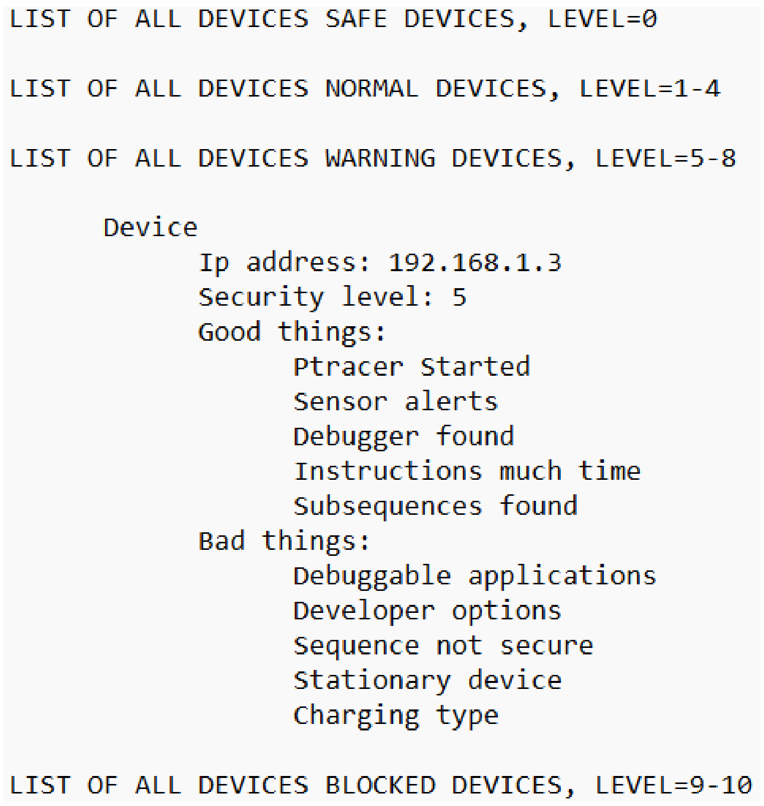
In order to be able to design and implement this safety mechanism for each device, an incremental scale of values was thought of starting from 0 which represents the absolute safety level of the device up to the value 10 which represents the maximum alert value for that particular device. For this reason it was also decided to divide these levels into intervals of different safety levels starting from the safe level with a single value of 0, then normal with a range of values 1-4, then warning with a range of values 5-8 and finally the level blocked with a range of values 9-10. With these range of values we are able to take certain future actions in relation to the security level of the device itself, such as whitelisting, blacklisting or continuous monitoring.

This security level model is very simple, in fact it is based on an incremental model where a unit is added to the security level for each control category that does not fall within a certain range of values determined in the training phase. The different types of control flags are listed below:

- Ptracer started: This flag verifies that the c++ library, called Ptracer, has been started correctly to be able to trace system calls since otherwise the low-level information could not be extracted.
- Sensor alerts: Represents the fact that during the execution of the application to be monitored, several times the device has assumed invalid positions such as the screen not visible to the user or the device upside down.
- Debugger found: This flag represents the most important one that verifies the presence of a debugger connected to the application that the user is using. In fact, if it is identified, the security level immediately passes to 10 which corresponds to the maximum alert state.
- Instructions much time: Represents the fact that while monitoring the device that the user is using, there are many system calls that have a duration greater than the maximum one captured during the training phase.

- Subsequences found: This flag tells us that during the monitoring phase many subsequences similar to the previous execution have been identified which consequently can identify an attacker who is analyzing the behavior of our application.
- Debuggable applications: This flag provides useful information regarding the possible presence of applications that are installed on the device to be monitored and that have the debuggable flag in the manifest file set to true.
- Developer options: This flag warns us about whether or not developer settings are active, usb or wireless debug options are a key factor in attaching a debugger.
- Sequence not secure: This flag warns us that while monitoring the execution of the application there have been many invalid sequences of syscall, or rather that they have not been traced during the training phase.
- Stationary device: This flag provides useful information that the device has assumed a stationary position during the execution phase which means that there is a possible attacker on a device that is debugging our application with a generic computer.
- Charging type: This last flag provides us with useful information regarding a possible connection with a generic computer if the type of recharging is the USB one.

An example of this level of security for devices that are connected to the web server can be seen in the figure below where we can immediately observe the devices in relation to the security range. In our case, only one warning level device appears to be connected as it has a security level between 5 and 8 and we can also observe what its suspicious aspects are as the presence of some applications with active debuggable flags, active developer settings, some sequences of system calls not mapped during the training phase, the stationary device positions many times and finally the charging type which is the USB one which implies a connection with a generic computer.



```
LIST OF ALL DEVICES SAFE DEVICES, LEVEL=0

LIST OF ALL DEVICES NORMAL DEVICES, LEVEL=1-4

LIST OF ALL DEVICES WARNING DEVICES, LEVEL=5-8

      Device
            Ip address: 192.168.1.3
            Security level: 5
            Good things:
                  Ptracer Started
                  Sensor alerts
                  Debugger found
                  Instructions much time
                  Subsequences found
            Bad things:
                  Debuggable applications
                  Developer options
                  Sequence not secure
                  Stationary device
                  Charging type

LIST OF ALL DEVICES BLOCKED DEVICES, LEVEL=9-10
```

Image 24: Device alert level

4.6. Considerations

In this chapter we have described what are the main points for identifying a possible debugger connected to the application that the user is using and at least the main points are the debuggable flag in the manifest file to be able to allow a debugger to connect, the developer settings enabled, the type of USB recharge as it is synonymous with a connection between the device and a generic computer, the motion sensor data in fact if the device is stationary it means that the attacker would be using a possible computer, the durations of system calls as a longer duration is a synonymous with a debugger connected because execution is slowed down or a breakpoint is fixed and the execution is stopped and finally the sequence of system calls to verify incremental executions as an attacker would be trying to study the behavior of the application itself.

It can be observed that there is no verified and reliable way to identify a possible debugger connected to the application that the user is using or an attacker's behavior pattern as they are only heuristics and statistics and above all this is a recent topic and there are not many scientific articles that we can read, study and implement any already tested algorithms. Consequently, my thesis work was to create a new high-level mechanism to implement a possible security level of a networked device.

To be able to implement this, I created a configuration file within the web server to be able to modify in relation to our objectives and requirements, the multiple variables that define the security levels and the necessary number of times that a particular event must occur before activating the flag of that specific section.

However, this model represents a high-level method for associating a security level to each device, in fact one of the future objectives of this tool consists in the implementation of machine learning algorithms in order to better train the model and have data most reliable for creating an ideal behavior of any Android application.

5. Validation

A fundamental chapter of this thesis consists in the validation of the architecture that has been described in the previous chapter. The control and validation process that we will see in this chapter consists of multiple phases including:

- The first phase where the test environment designed and developed to be able to test and validate the entire developed architecture including the two libraries and the web server is described
- The second phase consists of the training of the model created and used to identify an ideal behavior of an Android application through controlled executions of that particular application in order to then be able to observe any suspicious behavior because it was not tracked / stored in the training phase.
- The third phase consists of the one inherent to normal executions, i.e. all those executions that are performed by users to verify that the two imported libraries detect or not any incorrect / suspicious behavior based on the data collected and analyzed during the training phase and comparing them with the data collected during this last run.
- The fourth phase consists of that relating to executions of the abnormal type, i.e. all those executions which are carried out in order to verify that, when various suspicious factors occur during the current execution, the system functions correctly, i.e. detecting the alert flags and mapping between security levels and each device are performed correctly.
- Finally, the fifth phase consists of the one concerning the actions that can be taken when the system detects that a device has a high warning level and these can be, for example, disconnecting from the network, continuous monitoring and much more.

5.1. Test environments

In order to subsequently discuss the different phases that make up the whole process of checking and validating the architecture developed for this master thesis, I had to include the two libraries developed in some Android applications that people can use to demonstrate its possible real use.

I managed to include the two libraries in two different android applications. The first application consists of an audio recorder that allows using four buttons to start a recording, pause the recording, listen to the recorded audio and finally stop listening to the recorded audio while the second application consists of a simple open source calculator.

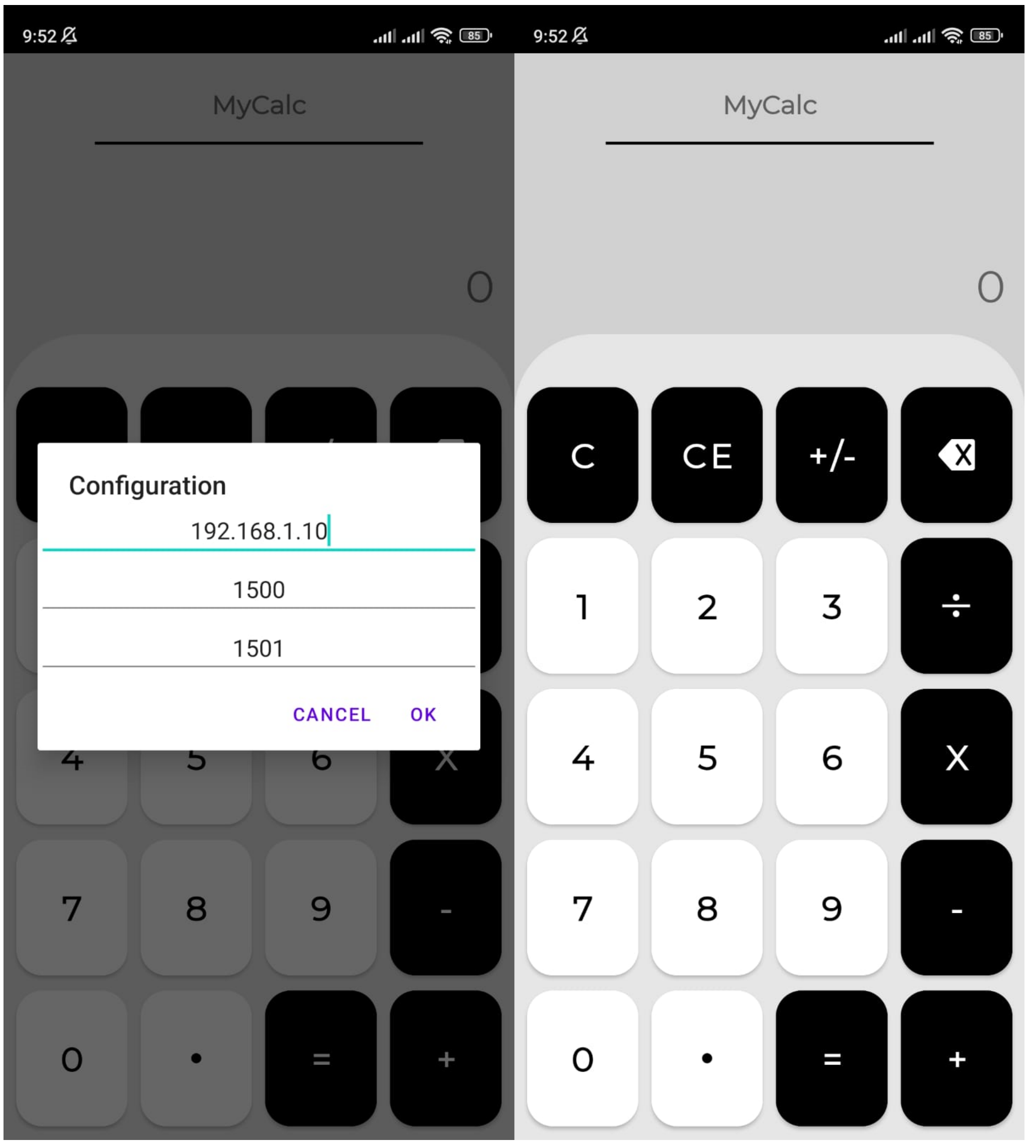
However, a future development consists in importing these two libraries also into more complex and consequently more used open source applications such as Mozilla Firefox or Vlc. In order to do this, an important development to do is a native C++ rewrite of all the high-level features present in this master thesis. This is necessary to be able to import all the functions in the same logical level and immediately above the kernel level in order to have immediate access to all the possible extracted data without any necessary modification of the code of the monitored application.

A point of fundamental importance consists in the list of permissions that are required to be able to run the application correctly and are as follows: the first permission consists of the permission necessary to be able to extract the list of all the applications that are installed on the device, the second permission is necessary to allow the application access to the internet used for the communication of the extracted data to the web server and finally the third permission which consists in the permission to be able to write information in memory outside the application and this is necessary for keep data in memory waiting to be sent to the web server.

Below we can see some screenshots of the Audio Recorder application, the calculator and the web server.

A fundamental point to discuss is the architecture of the entire developed project. The communication between application and web server consists of two separate tcp sockets, the first which is used for low level data communication and consequently used by the c++ library while the second used for high level data and consequently by the java library.

In fact, when an application with the libraries imported inside is started, a configuration message is shown where it is possible to insert the various data of the web server before being able to use the application such as the IP address of the web server and the relative ports for the two tcp sockets.

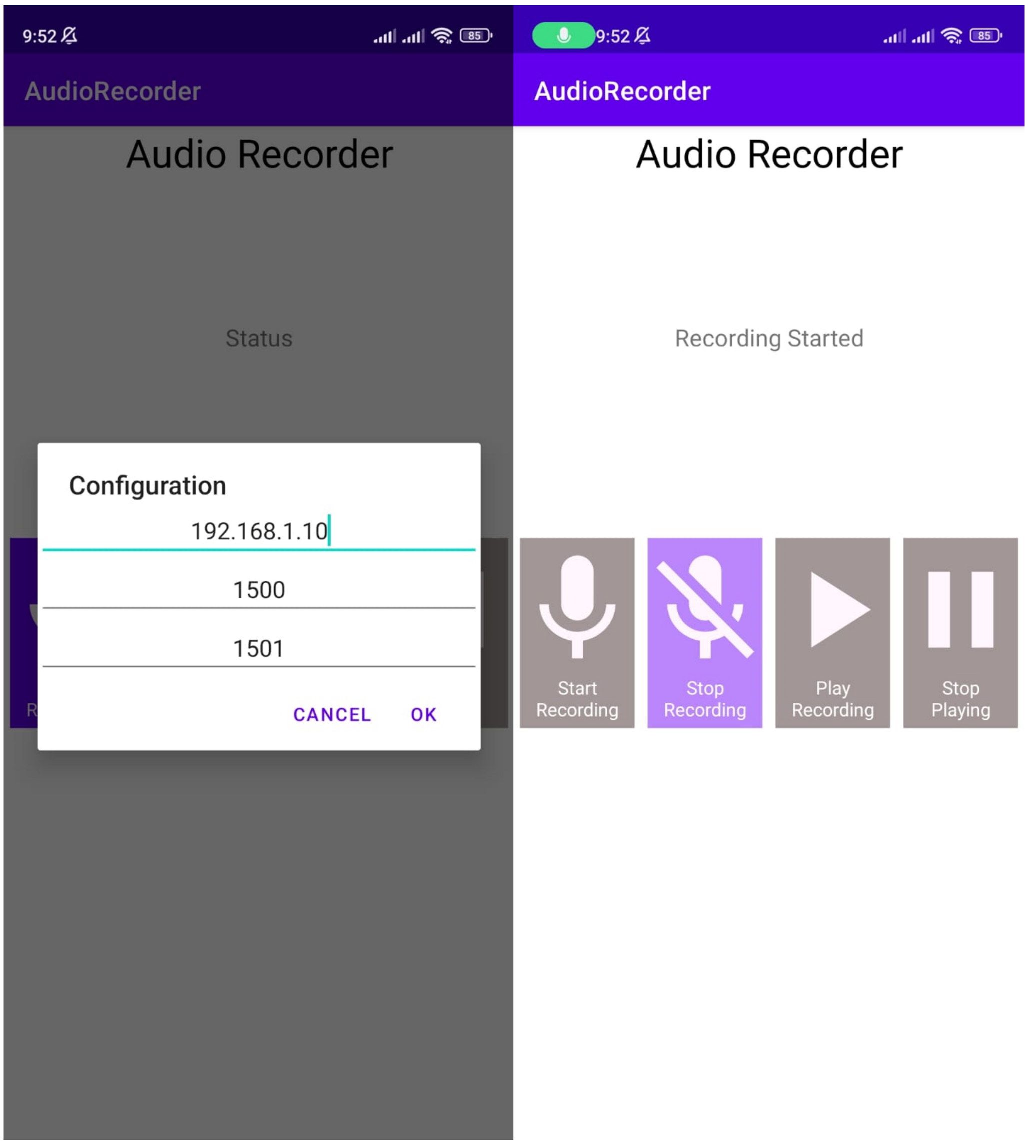

Configuration

192.168.1.10

1500

1501

Image 25: Calculator app



Configuration

192.168.1.10

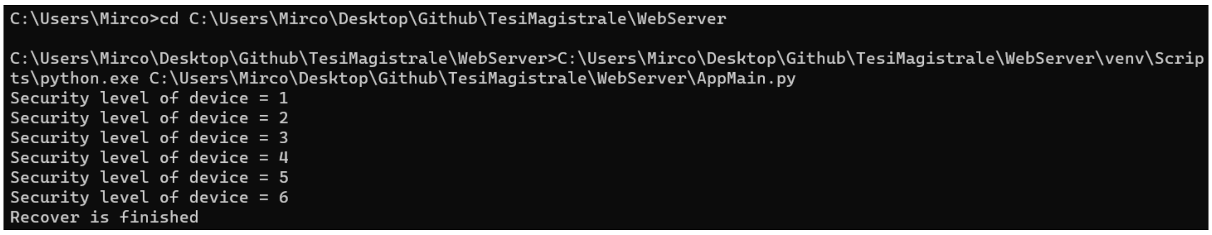
1500

1501

Image 26: Audio recorder app

Finally, one last screenshot concerns the web server, in this beta version it is a simple command line tool but a possible future development consists in the implementation of a management panel in the form of a web page where it will be easier to view the events that occur, more configuration options etc.

In the next image we can observe the functioning of the web server as a command line tool, specifically the two commands necessary for the correct execution. The first is necessary to enter the project directory while the second is necessary to start the entire web server. A key point consists in the recovery phase, i.e. the time necessary for the web server to open and read the contents of the various files where all the log data of previous executions have been saved, such as the list of previously connected devices, data relating to each system call, system call sequences, and more.



```
C:\Users\Mirco>cd C:\Users\Mirco\Desktop\Github\TesiMagistrale\WebServer

C:\Users\Mirco\Desktop\Github\TesiMagistrale\WebServer>C:\Users\Mirco\Desktop\Github\TesiMagistrale\WebServer\venv\Scrip
ts\python.exe C:\Users\Mirco\Desktop\Github\TesiMagistrale\WebServer\AppMain.py
Security level of device = 1
Security level of device = 2
Security level of device = 3
Security level of device = 4
Security level of device = 5
Security level of device = 6
Recover is finished
```

Image 27: Web server

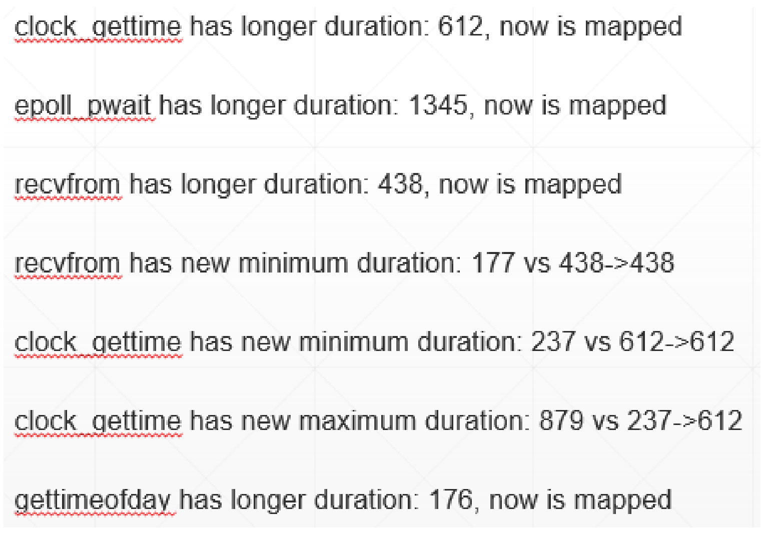
5.2. Training executions

A second fundamental point to be discussed in this chapter consists of the training phase, i.e. the phase in which we try to create a model that is as equivalent as possible to the behavior of a generic Android application. As we saw in the previous chapter, the data extracted and subsequently analyzed are divided into two types, the first for low-level data such as system calls, while the second for high-level data such as sensors, settings and recharge type etc.

As we can imagine, the first type of data, i.e. the low-level one, concerns system calls, their statistics, execution paths and finally their subsequences and all these types of information can be considered a good starting point to be able to create a good training model as we can identify the execution paths in relation to the degree of reliability of the device. Differently, on the second type of data, i.e. the high-level one, it is not possible to carry out a very reliable training phase as these data are different in relation to the user and his type of use as tracking data from a sensor to identify how much a device can move while the user is using it or how much a device is stationary over time are not key determining factors for the problem that we must solve.

Below we can see some examples of screenshots regarding the training phase of the model using all the data extracted with the low-level library.

The first type of information that we can use when running the web server in training mode consists of the characteristics concerning the system calls that occur when the user is running the application. An example can be seen in the figure below, two types of system calls can be observed, the first consisting of all those system calls that have not been found previously and consequently a message in the form " instruction now is mapped” while the second type consists of all those system calls that have been found previously and consequently a message appears in that specific line comparing the current duration with the duration interval of that specific system call.

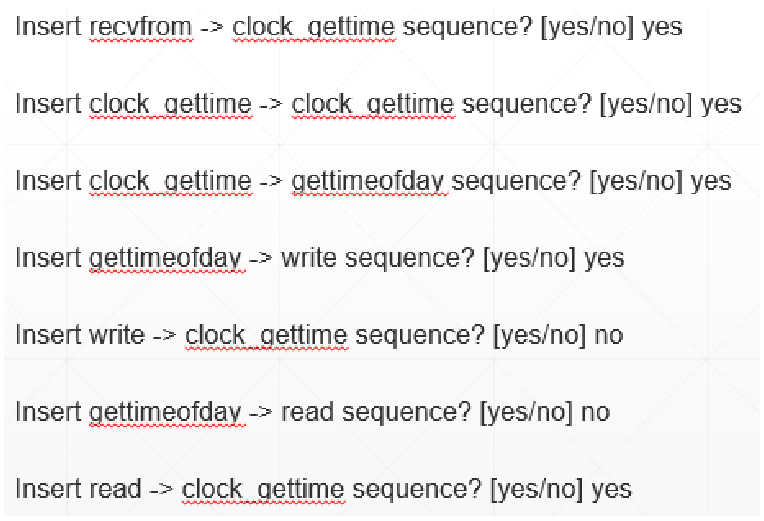


```
clock_gettime has longer duration: 612, now is mapped

epoll_pwait has longer duration: 1345, now is mapped

recvfrom has longer duration: 438, now is mapped

recvfrom has new minimum duration: 177 vs 438->438

clock_gettime has new minimum duration: 237 vs 612->612

clock_gettime has new maximum duration: 879 vs 237->612

gettimeofday has longer duration: 176, now is mapped
```

Image 28: Training of each syscall

A second type of information that is very useful in creating a model that is as reliable as possible consists in managing the possible sequences of system calls. In fact, in the following image we can see what the system calls can be after the current system call. This can be seen as a graph where each path indicates an execution path verified during the training phase. Consequently, a first point to be able to trace suspicious and non-suspicious executions consists in verifying the current path within this previously described graph and if this path is present then it means that it was traced during the training phase while if not mapped it means that it is a suspicious execution.

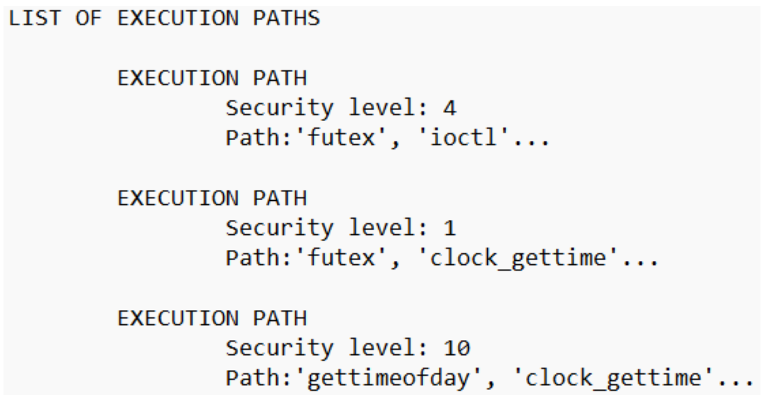
In order to implement this, it is impossible to automatically teach a computer which paths are valid and which paths are suspicious. As a result I thought of the following mechanism which we can see in the figure below. Whenever a sequence not yet mapped within this graph is detected, the administrator is asked whether this possible sequence is valid or not, in order to then proceed with the insertion in the valid sequences list or not.



```
Insert recvfrom -> clock_gettime sequence? [yes/no] yes

Insert clock_gettime -> clock_gettime sequence? [yes/no] yes

Insert clock_gettime -> gettimeofday sequence? [yes/no] yes

Insert gettimeofday -> write sequence? [yes/no] yes

Insert write -> clock_gettime sequence? [yes/no] no

Insert gettimeofday -> read sequence? [yes/no] no

Insert read -> clock_gettime sequence? [yes/no] yes
```

Image 29: Training of subsequences

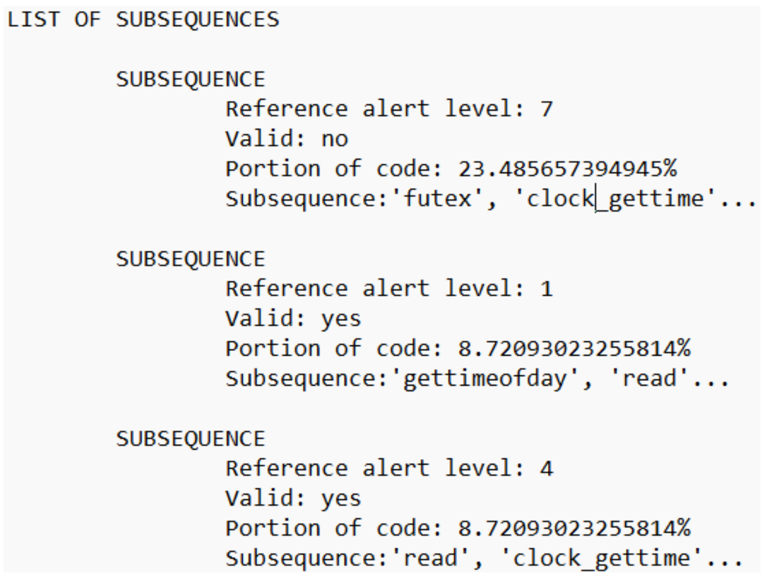
Finally, a last possible consideration in the training phase consists of some test executions where the sequence of system calls and the related security level are stored in each one in relation to what was described in the previous chapter. Consequently, below we can see an example of a log where three executions have been stored with the relative level of security.



```
LIST OF EXECUTION PATHS

        EXECUTION PATH
                Security level: 4
                Path:'futex', 'ioctl'...

        EXECUTION PATH
                Security level: 1
                Path:'futex', 'clock_gettime'...

        EXECUTION PATH
                Security level: 10
                Path:'gettimeofday', 'clock_gettime'...
```

Image 30: Training of execution paths

The next step of what was previously described consists of the check phase, i.e. the phase in which any subsequences are identified in relation to the saved executions. In fact, the figure below shows an example of subsequences that have been identified in the current execution in relation to the list of executions saved during the training phase. For each of these subsequences, we can observe that some information are managed including the security level associated with the execution in which the subsequence was found, the validity of the current subsequence being classified as valid or suspicious in relation to the range of the security level of actual execution details, how long the subsequence is in relation to the saved execution and finally the sequence of system calls present in the subsequence.



```
LIST OF SUBSEQUENCES

        SUBSEQUENCE
                Reference alert level: 7
                Valid: no
                Portion of code: 23.485657394945%
                Subsequence:'futex', 'clock_gettime'...

        SUBSEQUENCE
                Reference alert level: 1
                Valid: yes
                Portion of code: 8.72093023255814%
                Subsequence:'gettimeofday', 'read'...

        SUBSEQUENCE
                Reference alert level: 4
                Valid: yes
                Portion of code: 8.72093023255814%
                Subsequence:'read', 'clock_gettime'...
```

Image 31: Training of subsequences

5.3. Normal executions

The second execution mode consists of the check mode, i.e. the mode where the execution of the application that the user is using at the moment is verified using the data that were collected during the training phase in order to be able to compare and verify them later the real reliability of the current execution.

Below we can see a complete example of the logs that are generated in the web server divided into three parts as in order to be shown below they have been reduced by eliminating any similar lines in order to be able to attach all the potential functionalities of this project.

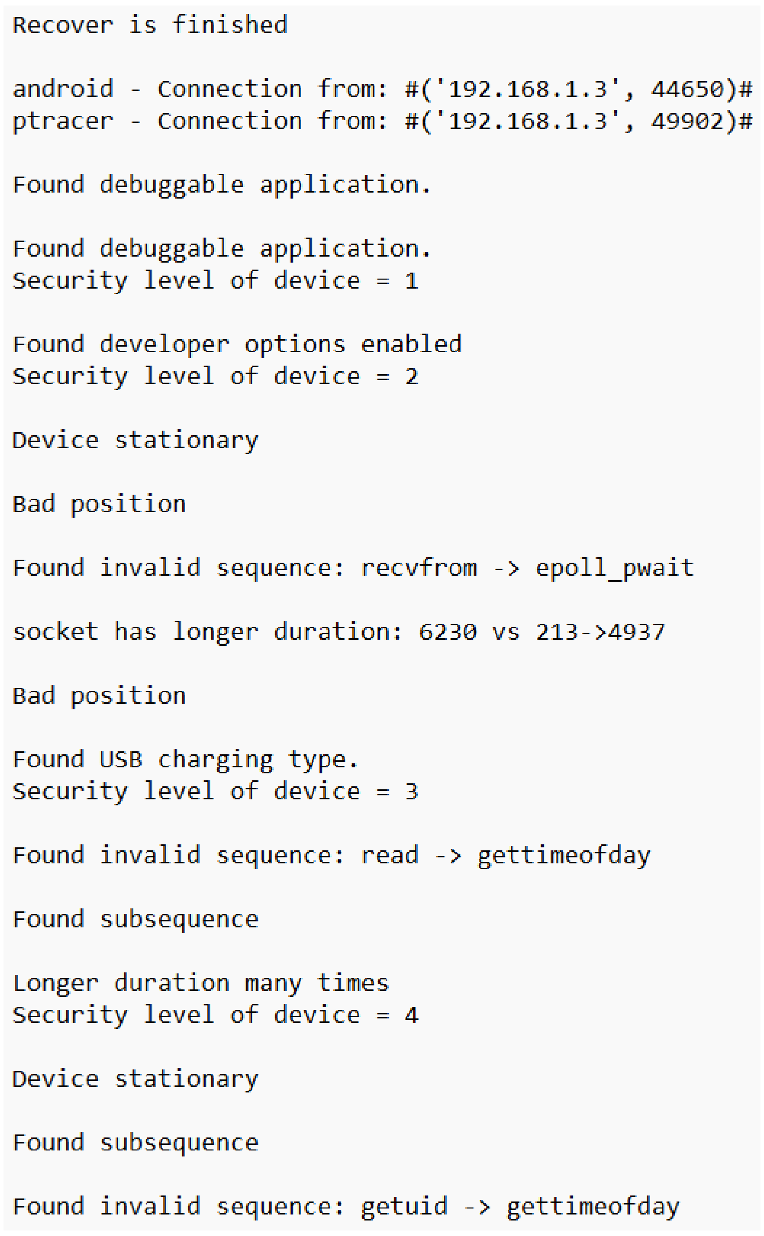
A fundamental point to be discussed consists in the possibility of modifying the tool in all its parts to one's liking and in relation to one's requirements. In fact, the settings file contains a lot of modifiable information starting for example from the address of the web server, the two ports necessary for communication with the two libraries, the paths in which to save the data of the current execution, any settings regarding the sequences, percentages, counters for the different flags and much more.

The first part of the log is characterized by some fundamental points including the recovery phase during which the data that was saved during the training phase is extracted and restored and the subsequent connection of the client to the web server via the two separate sockets each used from a specific library.

The second part of the log includes the same logic in all three following photos as we can observe what is happening in real time in the client and then eventually if something is suspected the security level of that particular client will be automatically increased. In fact we can observe several points within these logs:

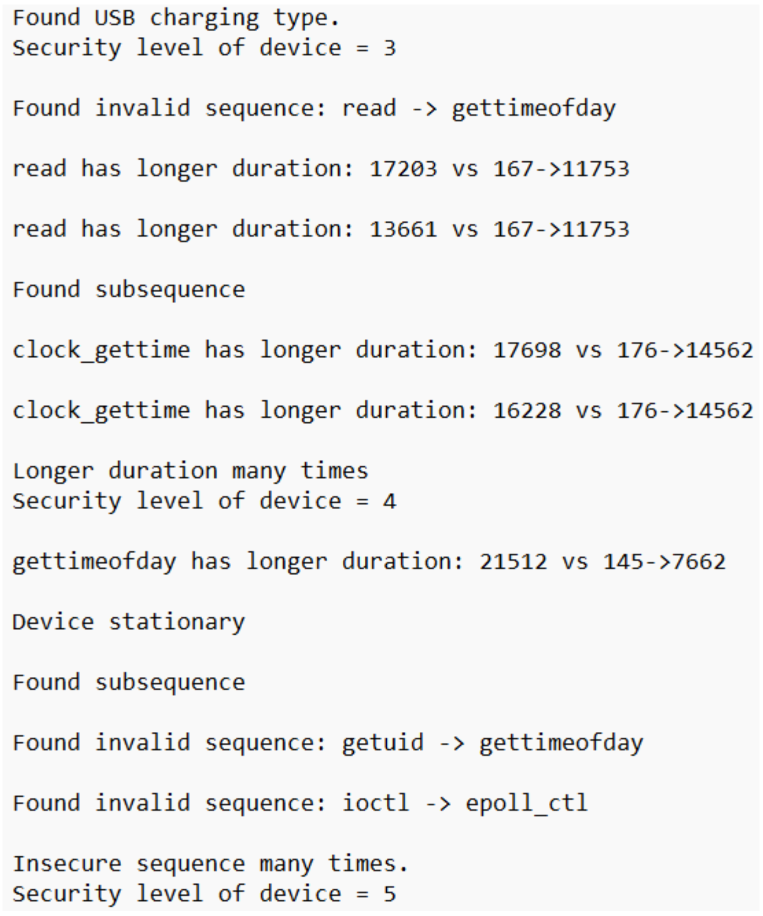
- The first point to analyze consists of debuggable applications, in fact the fourth line of the log shows an application installed on the device with this flag active in the manifest file. Consequently, immediately afterwards the security level of that device is set from level zero to level one and a message is also generated explaining the level change and this will happen for any variation of the security level.
- The second point is the active developer settings and the security level is increased again accordingly.
- The third point consists in identifying the moments in which the device is stationary, which and how many instructions have a longer duration, any subsequences present, any unmapped sequences and much more.
- Finally, the last two aspects to consider are the type of recharging of the device which provides us with very useful information regarding a possible connection to a generic computer and the identification of a possible connected real debugger.

Finally, in the third part of the log we can observe the closure of the two sockets used for each library imported into the application and finally the termination of the web server. This last step takes place only subsequently as the last operation that will be carried out will be the saving of all the logs relating to the last execution and the greater the execution time is, the greater the size of the saved logs will be.



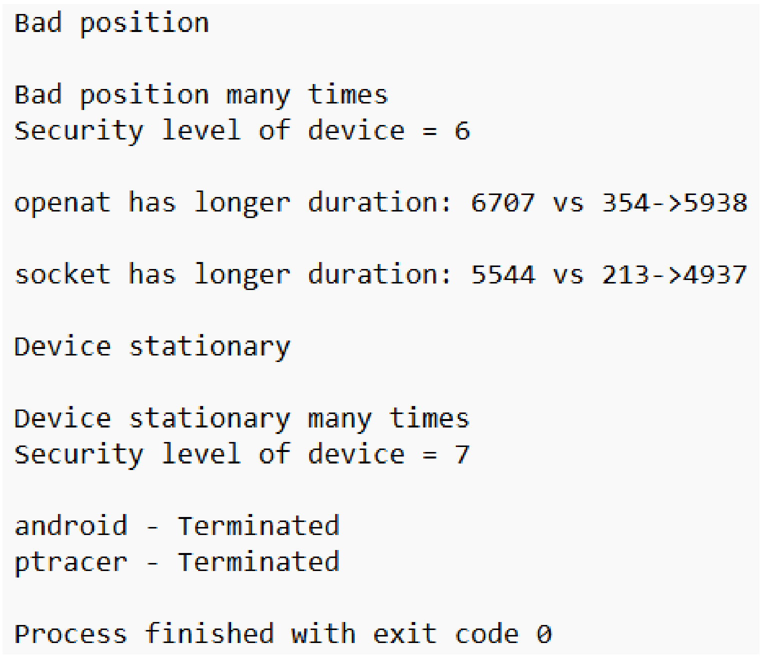
```
Recover is finished

android - Connection from: #('192.168.1.3', 44650)#
ptracer - Connection from: #('192.168.1.3', 49902)#

Found debuggable application.

Found debuggable application.
Security level of device = 1

Found developer options enabled
Security level of device = 2

Device stationary

Bad position

Found invalid sequence: recvfrom -> epoll_pwait

socket has longer duration: 6230 vs 213->4937

Bad position

Found USB charging type.
Security level of device = 3

Found invalid sequence: read -> gettimeofday

Found subsequence

Longer duration many times
Security level of device = 4

Device stationary

Found subsequence

Found invalid sequence: getuid -> gettimeofday
```

Image 32: First part of complete execution log



```
Found USB charging type.
Security level of device = 3

Found invalid sequence: read -> gettimeofday

read has longer duration: 17203 vs 167->11753

read has longer duration: 13661 vs 167->11753

Found subsequence

clock_gettime has longer duration: 17698 vs 176->14562

clock_gettime has longer duration: 16228 vs 176->14562

Longer duration many times
Security level of device = 4

gettimeofday has longer duration: 21512 vs 145->7662

Device stationary

Found subsequence

Found invalid sequence: getuid -> gettimeofday

Found invalid sequence: ioctl -> epoll_ctl

Insecure sequence many times.
Security level of device = 5
```

Image 33: Second part of complete execution log



```
Bad position

Bad position many times
Security level of device = 6

openat has longer duration: 6707 vs 354->5938

socket has longer duration: 5544 vs 213->4937

Device stationary

Device stationary many times
Security level of device = 7

android - Terminated
ptracer - Terminated

Process finished with exit code 0
```

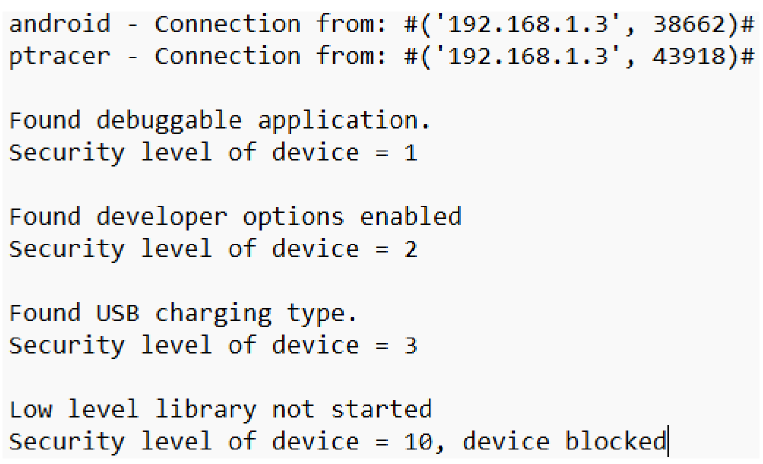
Image 34: Third part of complete execution log

5.4. Abnormal executions

During the validation and testing phase of this tool, some very interesting case studies emerged to analyze as they represent particular cases to be taken into consideration while monitoring the application that the user is using.

The first fundamental point to take into consideration consists in the management of the security levels which have been extensively discussed in the previous chapter. In order to be able to implement them at a high level, possible ranges for the values have been thought of, thus dividing everything into different macro levels as safe, warning and blocked.

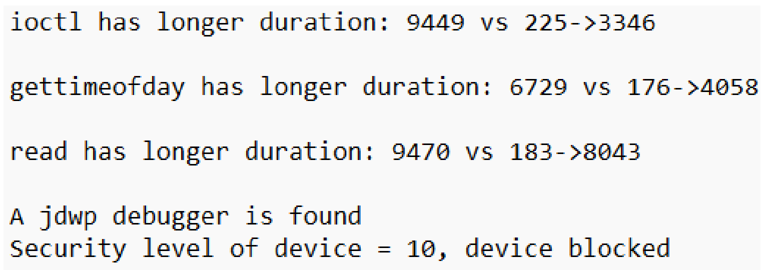
Consequently, another fundamental point to take into consideration consists of the two libraries that have been developed and that must necessarily be included in any Android application to be monitored. A first possible attack by an attacker consists in disabling a library which can be the low-level one or the high-level one indifferently. Consequently, when the web server identifies that there is no exchange of messages with any library, the security level of that particular device is increased to the maximum level, thus representing a blocked device as it is not possible to trace types of data needed for monitoring. Below we can see an example of a log where the low-level library has been disabled by the attacker.



```
android - Connection from: #('192.168.1.3', 38662)#
ptracer - Connection from: #('192.168.1.3', 43918)#

Found debuggable application.
Security level of device = 1

Found developer options enabled
Security level of device = 2

Found USB charging type.
Security level of device = 3

Low level library not started
Security level of device = 10, device blocked
```

Image 35: Ptracer not started

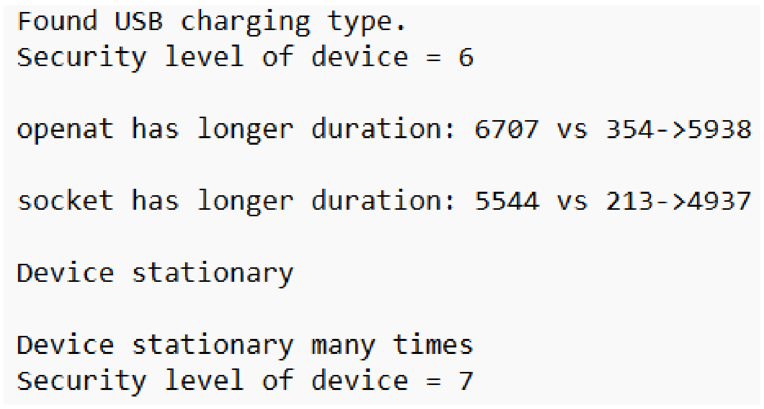
Another very interesting case study consists in the identification of a debugger connected to the application as the purpose of this thesis consists in the development of a mechanism with which it is possible to identify in advance a possible debugger by monitoring multiple types of data extracted from that particular device. Consequently, the libraries that I have developed, if they identify a connected physical debugger, increase the security level of that particular device to the maximum and in fact an example can be seen in the following figure.



```
ioctl has longer duration: 9449 vs 225->3346

gettimeofday has longer duration: 6729 vs 176->4058

read has longer duration: 9470 vs 183->8043

A jdwp debugger is found
Security level of device = 10, device blocked
```

Image 36: Debugger JDWP found

Another very interesting case study to take into consideration consists in observing the movement of the device in the long term and a possible type of USB recharge. This is because a possible attack by an attacker consists in connecting the device to a generic computer and then monitoring it remotely, making it remain in a stationary position over time.



```
Found USB charging type.
Security level of device = 6

openat has longer duration: 6707 vs 354->5938

socket has longer duration: 5544 vs 213->4937

Device stationary

Device stationary many times
Security level of device = 7
```

Image 37: Possible remote attack

Finally, a last point to take into consideration consists in the cases in which the device currently connected to the web server has a very high level of security and for this reason it is necessary to decide what actions to take. In fact, a future study regarding this tool, in addition to machine learning for data analysis, consists in the advanced management of connected devices to manage them by level or by whitelist and blacklist.

5.5. Automatic tests

A last point to be discussed consists in the creation of automated tests to be able to both verify the correct functioning of the designed and developed tool but above all to be able to automatically extract data which can then be used for a subsequent training phase for the inclusion of subsequent techniques of machine learning within this project.

Consequently, in order to implement this, we must remember that the information that we can extract from an Android device are of two types, the first at a low level which consists of system calls with their sequences, characteristics, duration, common subsequences, incremental executions and the second at a high level for sensor data, developer settings, charging type, device stationary and much more.

I thought of a hybrid solution as each library communicates autonomously to the web server using its reserved channel. Consequently, in order to do this, I used two different technologies for each type of information that can be extracted. To automate low-level information I used the MacroDroid Android application which allows you to create customizable macros to be able to launch applications, press on the screen, access the file manager, save the execution results in some variables and so on, all in a automatic and repeatable while for high-level data I created a fake java client that randomly communicates the different data in the same format as the library imported within the applications to be monitored.

Consequently, a development that I am carrying out in these weeks consists in the transformation of the high-level logs that have been discussed in the previous chapter in the form of a csv file in order to then be able to analyze them in a simpler way and thus be able to integrate them more easily with some future algorithms of machine learning.

I will also update this section over the next few weeks to cover the creation of the csv file and the mock java client that I will do.

6. Future developments

This chapter aims to explain what are the possible future developments of this tool that has been illustrated within this master thesis work. As can be seen from the structure of the chapter inherent to the solution designed to solve this problem, also in this chapter it is necessary to reason on the basis of the various components that make up the entire platform which are the high-level library, the low-level library and the web server.

6.1. JAVA library

The JAVA library consists of the high-level library or rather the extraction of data such as sensors, settings, type of recharge and the life cycle of the application from the application that you want to monitor. During the testing and validation phase we were able to observe some future developments which will be illustrated below.

6.1.1. Unique identifier of the device

The first future development consists in identifying the device where the Android application to be monitored is installed in a unique way. This for the moment consists only of an ip address that we know very well to be modifiable, variable, subject to potential attacks by an attacker, etc.

It turns out to be very convenient to be able to identify and monitor a specific device in the long term and consequently be able to find some detail, identifier or combination of data that is different for each device that can connect to the web server, it is necessary to subsequently bring this tool in an advanced stage of testing and production.

Consequently I thought of some data that can be used to uniquely identify each device and some examples can be the imei code, the mac address or a hash code obtained by coding multiple characteristics extracted from that particular device but some of this information cannot be extracted from this high level library due to some blocks of Google for privacy and security reasons.

6.1.2. Improve the tracing mechanism

Another point of fundamental importance to improve consists in the tracing mechanism of the multiple types of data that this library can extract. We have seen from previous chapters that there are many different pieces of information needed for the debugger detection problem and at the moment each of these is extracted from a different thread that remains running for as long as the application is running.

Consequently, a possible improvement consists in the possible redesign through the use of fewer threads or the use of some specially created particular data structures. Furthermore, a further development in this point consists in the creation of further settings for filtering, reduction of messages or time variables to decide how often to communicate for each type of information.

6.1.3. Improve the communication mechanism

A further point to be discussed regarding future developments consists in the communication mechanism of the multiple types of data that are extracted from the device. At this moment this communication takes place via a tcp socket shared with the web server. However we have seen that there are many different types of information and some of them such as sensors vary constantly.

Consequently, the optimization of the data encoding before sending it to the web server represents an important fundamental point in addition to memorizing them if during the execution of the application there should be disconnections from the network, closure of the socket, etc.

6.1.4. Reproject using native libraries

If we think back to everything that has been described in the previous chapters, we realize that in order to extract all the types of information useful for our problem, two different libraries are needed which must be important within the application to be monitored. Consequently, a first fundamental point consists in rewriting this high-level library into a low-level library perhaps written in C++ in order to be able to include it within the other library thus creating only one package necessary for monitoring.

Another point to discuss is the impossibility of extracting useful information to uniquely identify the device due to security and privacy issues limited by Google. Consequently a possible rewriting in native and consequently the different positioning of the library within the Android hierarchy would bypass this limitation.

6.2. C++ library

The second component of our infrastructure that we have developed consists of the low-level c++ library necessary for the extraction of system calls and all the information that can be extracted from them such as sequences, subsequences, characteristics of each syscall, details of any incremental executions and much more. Also this library, like all the other components of the infrastructure, has many future developments and these are described in the future developments chapter of Matteo De Giorgi's master thesis [8].

6.3. Web server

The last component of the infrastructure created to be able to solve the debugger detection problem consists of the web server which has the main task of receiving the multiple types of information extracted from the device, their subsequent analysis, the creation of the security level of that particular device and finally the decision of any actions to be taken.

6.3.1. Improve multi-client connections

The first future development of the web server consists in its redesign thinking about a possible interaction with multiple devices at the same time as for the moment only one device is supported at a time due to the too much amount of data that is extracted and communicated and the complexity of the structures data necessary for their storage and analysis.

Consequently, it is necessary to study and implement new efficient data structures due to the high number of stored information, the speed with which new information is stored and efficient algorithms for analyzing large amounts of data and different information.

6.3.2. Machine learning

A development that I am carrying out in these weeks consists in automating the testing and validation phase using MacroDroid and a fictitious JAVA client as previously described and I am also trying to transform the high-level information into a csv file which will subsequently be used to train some machine learning algorithms.

6.3.3. Next action

A fundamental point to plan concerns the actions to be undertaken after calculating the level of security of a given device connected to the web server and this serves to be able to monitor it on a long-term basis in a specific environment in relation to its level of security. For the moment the security levels, as previously described, are divided into macro levels where for each one we can initially think of a whitelist, blacklist or greylist in relation to different levels of security.

6.3.4. Web UI

One last future development of the web server consists of a web page style redesign in order to allow a better view of everything that is happening to the different Android devices to be monitored. This is because the current version of the web server consists of a command line tool and consequently any operation or information we want to display must be printed in the output stream, making the entire output increasingly messy and complex to understand.

Consequently, its redesign thinking a web page divided into modules would represent the best solution and for this, each module represents a type of information to be shown and it is also possible to set some possible interaction with the different modules to be able to have greater detail of them.

7. Conclusions

With this thesis I wanted to design and develop this remote debugging detection tool in Android as this represents a new problem, there is no tool capable of verifying, by analyzing multiple types of data, if a possible debugger is connected to the application or if an attacker is trying to understand the functioning of the behavior of the application that we want to monitor using for example incremental executions.

This thesis can be understood as a continuation of that of another classmate since he had developed the low-level library to be able to extract the system calls, the relative sequences and a possible finite state automaton to show the possible execution paths of the application in question.

Consequently the work of this thesis was to develop two main points. The first consists of a high-level library, which will be included together with the other library, and will be used to be able to extract further data such as sensors, settings, charging type, applications that can be connected to a debugger and the lifecycle of that specific application while the second point consists of a web server necessary for both the analysis of all this data collected and for subsequent management in order to be able to observe what happened during the execution of that specific application.

However, this thesis discusses a high-level heuristic project, in fact the model created by me for the management of the security level of each device connected to the web server represents the simplest way to be able to develop it as through an initial study of the macro areas that can identify an attached debugger, and then a flag was created for each of these areas to identify valid and invalid ones for each monitored execution. In our testing and validation phase each area had the same weight making sure that the security level belonged to the range 0-10 but this can be changed within the project.

A very interesting future development that we are carrying out consists in the inclusion of machine learning algorithms to automatically analyze if an execution is valid or not. In fact, in this period I am trying to encode the information contained in the logs at a high level format within a csv file to be able to parse it more easily.

Bibliography

1. Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. pages 15–26, 10 2011. ISBN 9781450310000. doi: 10.1145/2046614.2046619.

2. Alessandro Cabutto, Paolo Falcarin, Bert Abrath, Bart Coppens, and Bjorn De Sutter. Software protection with code mobility. pages 95–103, 10 2015. doi: 10.1145/2808475. 2808481.

3. Abhishek Chaturvedi, Sandeep Bhatkar, and Ramachandran Sekar. Improving attack detection in host-based ids by learning properties of system call arguments. 01 2005.

4. Gianluca Dini, Fabio Martinelli, Andrea Saracino, and Daniele Sgandurra. Madam: A multi-level anomaly detector for android malware. In Mathematical Methods, Models, and Architectures for Network Security Systems, 2012.

5. H.H. Feng, O.M. Kolesnikov, P. Fogla, W. Lee, and Weibo Gong. Anomaly detection using call stack information. In 2003 Symposium on Security and Privacy, 2003., pages 62–75, 2003. doi: 10.1109/SECPRI.2003.1199328.

6. Steven A Hofmeyr, Stephanie Forrest, and Anil Somayaji. Intrusion detection using sequences of system calls. Journal of computer security, 6(3):151–180, 1998.

7. Zhen Liu, S.M. Bridges, and R.B. Vaughn. Combining static analysis and dynamic learning to build accurate intrusion detection models. In Third IEEE International Workshop on Information Assurance (IWIA'05), pages 164–177, 2005. doi: 10.1109/IWIA.2005.6.

8. De Giorgi Matteo. System Calls Monitoring in Android: An Approach to Detect Debuggers, Anomalies and Privacy Issues. Master's degree, Università Ca' Foscari Venezia, 2022.

9. Yuan Zhang, Min Yang, Bingquan Xu, Zhemin Yang, Guofei Gu, Peng Ning, X. Sean Wang, and Binyu Zang. Vetting undesirable behaviors in android

apps with permission use analysis. In Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS ’13, page 611–622, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450324779. doi: 10.1145/2508859.2516689. URL https://doi.org/10.1145/2508859.2516689.

10. Rubin Xu, Hassen Saïdi, and Ross J. Anderson. Aurasium: Practical policy enforcement for android applications. In USENIX Security Symposium, 2012.

11. M. Ceccato, P. Tonella, C. Basile, P. Falcarin, M. Torchiano, B. Coppens, B. De Sutter, Understanding the behaviour of hackers while performing attack tasks in a professional setting and in a public challenge, Empirical Software Engineering 24 (1) (2019) 240–286. doi: 10.1007/s10664-018-9625-6. URL https://doi.org/10.1007/s10664-018-9625-6

12. T. O. Foundation, Owasp mobile security testing guide (2018). URL https://mobile-security.gitbook.io/mobile-security-testing-guide/

13. B. Abrath, B. Coppens, S. Volckaert, J. Wijnant, B. De Sutter, Tightly-coupled self-debugging software protection, in: Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, SSPREW ’16, ACM, New York, NY, USA, 2016, pp. 7:1–7:10. doi:10.1145/3015135.3015142

14. P. Faruki, H. Fereidooni, V. Laxmi, M. Conti, M. Gaur, Android code protection via obfuscation techniques: past, present and future directions, CoRR abs/1611.10231.

15. S. Banescu, A. Pretschner, A tutorial on software obfuscation, in: Advances in Computers, Vol. 108, Elsevier, 2018, pp. 283–353

16. J. Hoffmann, T. Rytilahti, D. Maiorca, M. Winandy, G. Giacinto, T. Holz, Evaluating analysis tools for android apps: Status quo and robustness against obfuscation, in: Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy, 2016, pp. 139–141.

17. J. Park, H. Kim, Y. Jeong, S.-j. Cho, S. Han, M. Park, Effects of code obfuscation on android app similarity analysis., JoWUA 6 (4) (2015) 86– 98.

18. J. Wan, M. Zulkernine, C. Liem, A dynamic app anti-debugging approach on android art runtime, 2018, pp. 560–567. doi:10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00105.

19. Y. Jing, Z. Zhao, G.-J. Ahn, H. Hu, Morpheus: automatically generating heuristics to detect Android emulators, in: Proceedings of the 30th Annual Computer Security Applications Conference on - ACSAC '14, ACM Press, New Orleans, Louisiana, 2014, pp. 216–225. doi: 10.1145/2664243.2664250. URL http://dl.acm.org/citation.cfm?doid=2664243.2664250

20. K. Lim, Y. Jeong, S. je Cho, M. Park, S. Han, An Android Application Protection Scheme against Dynamic Reverse Engineering Attacks, Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications 7 (3) (2016) 40–52. doi:10.22667/JOWUA. 2016.09.31.040. URL https://doi.org/10.22667/JOWUA.2016.09.31.040