VU University Amsterdam
Faculty of Sciences
Department of Computer Sciences

Bachelor thesis

Porting TraceDroid to Android 4.4

Dimitri Diomaiuta

supervisor
Victor van der Veen

July, 2017

Abstract

The Android operating system, with 2 billion devices monthly active and an estimated market share over 80%, is growing faster than the expectations. Together with the growth of the number of devices that use Android there is an increase of malware directed to the most used mobile operating system. In 2016 over 8 billions installations of malicious packages have been detected.

The cyber security community has been aware of these malware since the early days of Android. Various frameworks were created to provide comprehensive analysis of Android application to detect suspicious and malicious code. The TraceDroid Analysis Platform is a scalable and automated framework providing both static and dynamic analysis by means of integrating Androguard and an Android sandbox.

This sandbox, dubbed TraceDroid, is the engine providing the dynamic analysis functionalities. It works by modifying a default Android operating system to enable profiling and consequently tracing the entire flow of an application. Every Android version has its own API level, which determines the compatibility with an application. The aim of this research is to update the Android sandbox, namely TraceDroid, from version 2.3 to version 4.4 to increase the compatibility rate.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Android, with over 2 billion devices monthly active (as of May 2017), is the most used mobile operating system [3]. Google choice of designing an operating system both open source and compatible with different hardware made it an attractive solution for smartphone manufacturers [6], contributing in increasing its mobile OS market share up to 86% in the third quarter of 2016 [7]. One of the key components of Google's operating system is that it is easy for the developers to use the platform [6], as a consequence the increasing number of users is directly proportional to the increasing number of application programmers. This has led to 2,800,000 apps available and more than 65 billion cumulative downloads since 2010 [8, 9].

The growth of this platform did not come without drawbacks. It, in fact, became the target of malware with the aim of stealing data or money from the users. Kaspersky lab [2] detected in 2016 over 8 million malicious installation packages, showing that Android security is at risk. In late May 2017, as confirmation of this, a malware, named *Judy*, was discovered in circa 50 applications in the Play Store [5]. The apps containing the malicious code have been downloaded approximately 36 million times and have been responsible for generating fraudulent clicks to gain money from adverts. Another example of a recent (April 2017) malware available directly from the Play Store is *Dvmap* [11]. It is a sophisticated trojan that not only installs its modules into the system but it also injects the code into the system runtime libraries. 50,000 times is the estimation of the malicious program total downloads.

Both Google and the scientific community have acknowledged the severity of the situation. Google, recently, developed a new system, dubbed Play Protect, actively scanning Android devices and new versions of apps uploaded to the Play Store to perform malware detection [4]. The system is new (May 2017), and aims at being a bigger security framework including safe browsing protection, and at locating and locking the device in case it has been lost. The scientific community, on the other hand, is aware of this situation since the early days of Android. Many tools have been developed for static and dynamic analysis. Static analysis approaches focus on analyzing the source code or the byte code of an application, the Androguard framework is an example of a tool performing this [12]. Dynamic analysis, on the contrary, focus on analyzing a program when it is running. Andrubis, Forsafe, Joe Sandbox Mobile and TraceDroid are examples of sandboxes performing dynamic analysis [10].

We present, in this paper, an update of the TraceDroid sandbox in order to increase the compatibility with Android applications [1]. This sandbox is the dynamic analysis core of a larger automated framework, dubbed TraceDroid Analysis Platform (TAP) that performs both dynamic and static analysis. TraceDroid is a sandbox based on Android 2.3.4 that works by modifying the core files of the Dalvik Virtual Machine to enable profiling and tracing of the execution of an application. The main goal of the update is to port it to Android 4.4: the last Android version to have the Dalvik Virtual Machine as the default runtime environment.

The document is structured as follows. In Chapter 2 we explain the main concepts about the Android architecture useful at understanding the design and implementation of the system being ported. Chapter 3 discusses the framework design and in Chapter 4 we discuss the implementation and limitations of our work. In Chapter 5 we provide an evaluation of the system aimed at checking the increase of compatibility with Android applications. Finally, we consider future work related to Android 7 and general concluding notes in Chapter 6.

Chapter 2

Background Information

The TraceDroid sandbox enables profiling of Android applications by modifying the Dalvik VM of Android [1]. It is important, hence, to understand how Android works, how its system architecture is designed and especially how the virtual machine running the applications is structured. This chapter contains a brief overview of the whole architecture and a detailed explanation of the runtime core, namely the DVM.

2.1 Android System Architecture

The 5 layers that compose Android at the software level are shown in Figure 2.1[1]. Below there is a brief overview of the Android system architecture [13]:

1. The bottom layer is the *Linux Kernel*, a slightly modified version of the standard one. The improvements are aimed at adapting the kernel to the hardware constraints of the mobile devices.
2. The libraries layer is a set of C/C++ essential C libraries rewritten to support the ARM architecture. These libraries are accessible from the application framework.
3. The runtime component is based on the dalvik virtual machine, with its specific libraries, and a collection of core libraries, specifically the interoperability libraries of the Java programming language. Since version 4.4 (codename Kitkat) it is written in C++.
4. The Application Framework, written in Java and interpreted by the DVM, is the main layer of the Android framework and provides a rich set of components used by the applications to access the device functionalities. It contains also background processes that manage running applications and interoperability between them.
5. The last and higher layer is the Applications layer. Applications are built on top of the Applications Framework and provide the interfaces end users interact with. Most of the apps are written in Java but developers are also free to integrate it with native code for different purposes (e.g. games that extensively use GPU).

[1] via:`http://en.wikipedia.org/wiki/Android_(operating_system)#Linux`

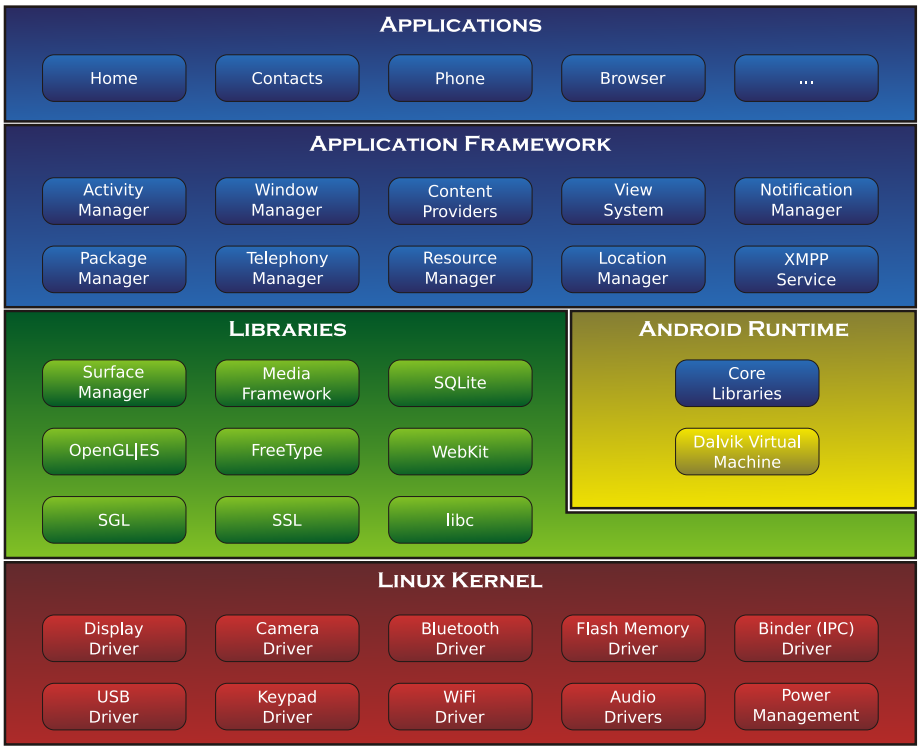

Figure 2.1: Android low level system architecture

2.2 Dalvik Virtual Machine

Google decided to adopt Java as the programming language to write Android applications due to its popularity and pervasive nature. The Java bytecode, produced by the language, is interpreted and executed by a virtual machine. A Java mobile ecosystem already existed, named JME, but because of its fragmentation Google has created its own suite of tools and its own VM, called Dalvik Virtual Machine [14].

Google implemented this VM with the awareness of the hardware constraints of the devices running their operating system. Smartphones, in fact, have a limited processor speed, limited RAM size, no swap space and are battery-powered. The runtime core, nevertheless, is based on modern OS principles and process isolation is granted by having each application running in its own instance of the DVM. This, furthermore, improves also security by virtue of the virtual machine sandboxed nature.

Android OS developers, at the bytecode level, performed also an optimization by adopting a different format with the primary goal of preserving memory. The new format, called DEX (Dalvik EXecutable) bytecode, differs from the standard bytecode since Java `.class` files, obtained after compiling the sources, are converted into a single `.dex` file using the `dx` utility. On one hand the standard `.class` files have their private heterogeneous constant pool for storing different types of constants, on the other the `.dex` files have different type specific constant pools shared among all the classes of the same program. This

constant part accounts for a big part of the Java class files, circa 61%. Space is, hence, saved by having extra pointers referring to the same constants in the shared constant pools.

One of the fundamental components of the DVM is the *Zygote* process. This special VM process is started at boot and is the parent of all the future spawned VM. Once started, it initializes the core libraries and listens for the request of a new application. In this case the `fork()` command is called and a new VM instance is started. Forking a VM allows to share the core libraries and minimize the startup time. Each Application, as a consequence, runs in its own isolated process which has a unique `uid` (user identifier) that can only access the set of features, verified during installation, requested in its manifest.xml file. This design enforces not only minimized load times but also increases the whole OS security.

Another difference between the JVM and Google implementation is that the architecture is not Stack-based but register-based. Standard virtual machines are based on Stack data structures because of its simplicity. The DVM, in contrast, that is based on registers, lacks code density (circa 25% larger source code) but increases the overall performance by about 32%. Having in mind all the hardware constraints of the devices the Android OS is designed for, this is an acceptable compromise.

Chapter 3

Design

The TraceDroid sandbox is the core of the dynamic analysis provided by the TAP framework [1]. We present, in this section, the aim of this modified Android OS and its main design points. This overview is fundamental in understanding both how the program works and how the porting process, discussed in the next chapter, has been achieved.

3.1 What to collect

The main goal of the sandbox is to perform dynamic analysis of Android application and to provide extensive and relevant traces of an executed application. The output should provide an overview of the app behavior and its control flows. Cyber security experts can, afterwards, analyze the obtained results to understand whether the analyzed code is malicious and, hence, be classified as malware.

Android applications are written in Java programming language and after been compiled in `.dex` file format they are executed by the Dalvik Virtual Machine that interprets the code. This intermediate runtime layer, described extensively in the previous section, is where the core of dynamic analysis resides. Here the modified version of the VM, dubbed TraceDroid, traces all the relevant data of a running process. The profile obtained should contain specific information like the names of the method called, the method's class descriptor, the type and value of parameters and returning values and the timestamps of the computations.

3.2 Specification

TraceDroid aims at extending the already default Android profiling feature in giving more accurate and precise information about methods execution and process flow. The traces should be as close as possible to the Java source code of the .dex files running in the DVM. The following are the main requirements that the dynamic analysis program should implement:

- A common interface that specifies which app should be analyzed at runtime.

- A common interface to store the output in files which are named in such a way to understand the process and the thread the traces are generated from.
- Providing full automation through its interfaces in order to be plugged in a bigger framework such as TAP.
- The traces should consist of the following:
  - Timestamps of when each computation takes place.
  - Called method name.
  - Called method class descriptor.
  - Called method return type and modifiers.
  - Values and types of the parameters and return values.
  - Thrown exceptions unwinding.
  - Indentation of the calls in order to make it easier to understand the results.

3.3 Interaction

In this section we describe how a user, or a program, should be able to interact with the TraceDroid image in order to obtain traces of a running app. First the Android SDK needs to be installed on the machine where we want to run the sandbox. After installation an Android Virtual Device (AVD) has to be created: it will work as a container with all the properties necessary for an Android system image, in this case the custom TraceDroid one, to be emulated. We can then start the emulator and interact with the modified Android sandbox via the `adb` (Android Debug Bridge) command line interface.

Figure 3.1 shows how to interact with the system. The actor is depicted as a user but it can also be automated in order to plug the custom image into a bigger framework like TAP. After starting the emulator the user interacts with the system with the command line interface provided by the adb tool. In the case we want to trace an app that is not installed in the AVD we first need to install it with adb. This is not shown in the sequence diagram for simplicity reasons.

The user, or the program that uses the dynamic analysis sandbox, locates the unique identifier (`uid`) of the application that wants to profile and writes it into a file, named uid, that pushes to the `/sdcard/` of the AVD via adb. The actor, after this, chooses between the activities that the application offers and starts one of them. All the activities should be started in order to have more comprehensive and detailed dynamic analysis of the process, this is exactly what TAP does. The activity is then started by sending the command to `adb`.

The system now gathers control and adb sends the app's activity invocation to the *Activity Manager*. The AM, that is a process-like manager, forwards the intend of starting a new activity to the *Zygote* process. The *Zygote*, the parent of all the VM, invokes a `fork()` system call to create a new virtual machine that performs as a container of the application whose activity is started. The functions in the `Init` file create a new VM. These functions are modified in the TraceDroid version in order to check whether the uid of the process started is the same as the one stored in `/sdcard/uid`. Depending on whether the uid is the same or not we have two different flows:

1. The uid is the same: the activity is initiated and the modified `Profile` class is activated to filter all the calls made by the process. The traces are then written to files under `/sdcard/`. When the activity finishes the VM returns, without killing the process, and the user can pull the dynamic analysis results from the `/sdcard/` via `adb`.
2. The uid is not the same: the activity is initiated but the `Profile` class is not activated since the user did not specify to trace this process. In this case the new VM completes the requested activity and returns to the AM once it finishes.

This design respects the requirements points specified in the previous section. The described sequence provides both a common interface for tracing an app (the push uid `/sdcard/`) and for obtaining the dynamic analysis results (the pull `/sdcard/traces*`). The system, furthermore, allows for full automation using the `adb` command line tool.

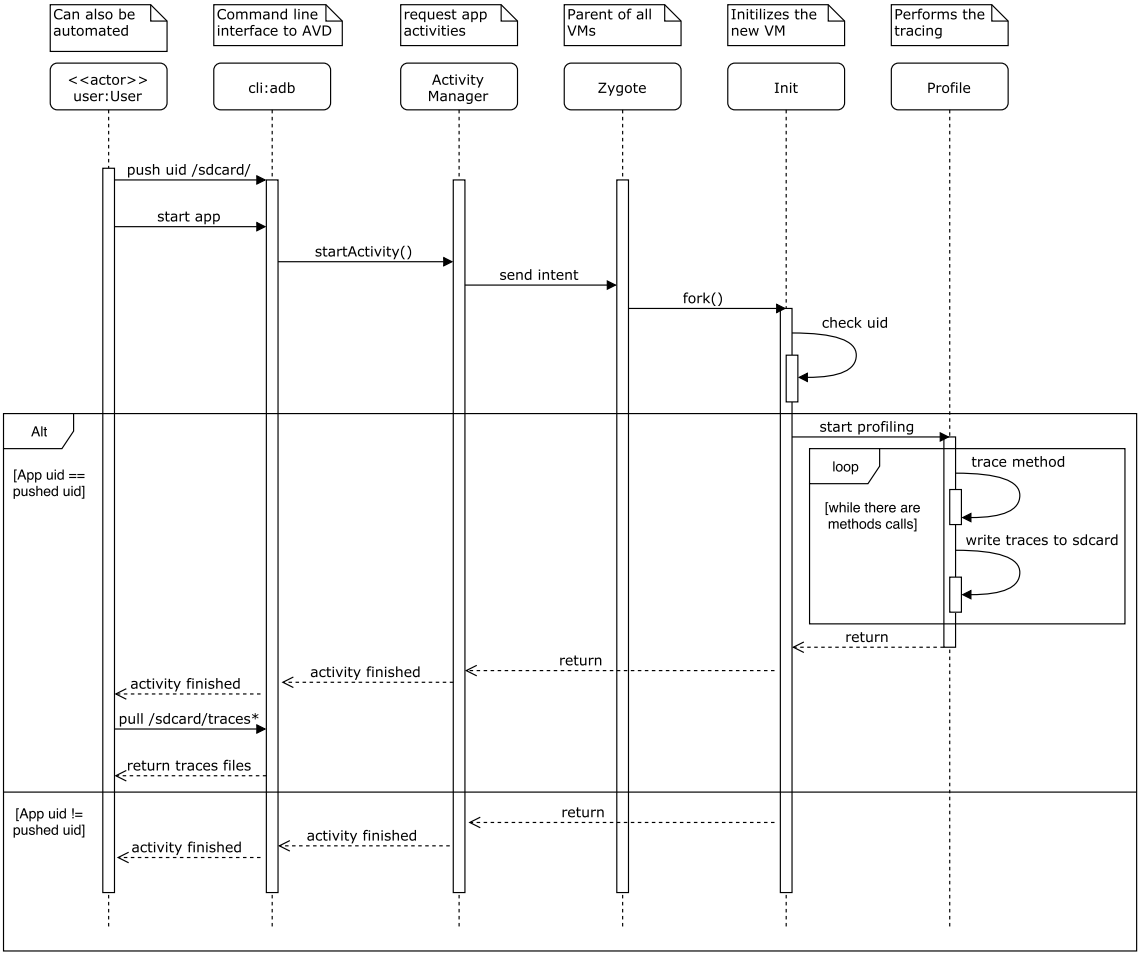

Figure 3.1: Interaction Sequence Diagram

Chapter 4

Implementation

In this chapter we discuss the implementation of the porting process: updating TraceDroid from Android version 2.3.4 (codename Gingerbread) to Android version 4.4 (codename KitKat). The whole process is reproducible, the technical guides and sources can be found on the Github page of the project [15].

4.1 Porting Process

4.1.1 Target version

The original version of the sandbox is based on Android 2.3.4, one of the earliest versions of the Google OS. Nowadays Gingerbread is largely outdated and accounts for only 0.8% in the Android platforms distribution [16]. The main drawback of having an outdated version at the core of dynamic analysis is that the API level used by the sandbox is not supporting most of the applications. Applications, in fact, contain in their manifest file an integer value called *minSdkVersion* which expresses the minimum API level required to run the application [17]. Developers, as a consequence, are not interested in creating apps compatible with an Android version which has just a 0.8% of market share. The TraceDroid compatibility rate, as a consequence, decreases drastically.

We, hence, decided to port the system to an higher Android version to increase the compatibility rate with applications to be dynamically analyzed. We have chosen Android KitKat as the target version (Android 4.4, API level 19). The main reason for this choice is that version 4.4 is the last one still using the Dalvik Virtual Machine as default runtime environment. In KitKat the VM is now implemented in C++ instead of C but the file structure and the functions provided are similar, making the porting process smoother. Furthermore, the market share of this platform is a high 18.1% and it is the third most used Android version. Developers, being aware of the data, are interested in supporting this major version and hence most applications will have at least version 19 or lower as *minSdkVersion* in their manifest file. As a consequence porting TraceDroid to Android 4.4, despite not being the latest Android version, will increase the compatibility rate significantly.

4.1.2 Porting tools

The first step into porting TraceDroid was to setup a build environment to download the Android source tree from Google git repository and to compile it to obtain a default system image. The source code can be compiled only under Linux or Mac OS. On top of this Google suggests to use Ubuntu 64-bit as the default Linux distribution to build Android [18]. We chose to use a docker container as a build environment for Android 4.4 to avoid dependencies issues on the client Linux machine we were working from. We selected Ubuntu 12.04 64-bit image from the official Ubuntu docker repository because of its compatibility with older versions of the Google mobile OS. We then downloaded the Android source code [19] and successfully compiled it after solving the dependencies issues causing compilation errors. The technical details regarding the dependencies installed in the docker container for compiling both Gingerbread and KitKat can be found in the Github repository of the project [15].

We needed to install also the Android SDK tools [21] to be able to emulate an Android system image. We performed this by installing Android studio and the required dependencies in our client Linux machine, in the specific *debian 3.16.0-4-amd64* [20]. The SDK tools allow us to create an Android Virtual Device (AVD) to hold the system image with its properties and then emulate it [15]. This is fundamental to access our custom TraceDroid system image and interact with it via the `adb` (Android Debug Bridge) command line interface [22].

4.1.3 Porting workflow

TraceDroid [1] performs the tracing by modifying the behavior of the virtual machine containing the application that we want to analyze dynamically. The changes, hence, involve the files under the `dalvik/vm/` folder in the Android source code. The first stage in the porting process was to understand the difference between the original TraceDroid `dalvik/vm/` files and the original ones of Android 2.3.4, the Android version on which it is based. We used the `diff` command recursively between the two folders to keep track of all the differences at the source code level.

The second stage was to port the changes step by step to the `dalvik/vm/` source files of Android 4.4. The modifications were ported together with the addition of debugging messages to the Android logs via the `ALOGD` macro. After performing every step in the porting process we pushed the implemented files to the docker image responsible of building the sources. We compiled the `dalvik/vm/` module with the mm utility and then linked it to the other modules with a make at the root of the Android source tree. After the compilation was terminated we pulled the generated system image from the container and we emulated it. We started some activities to trigger TraceDroid and analyzed the logs after it was terminated. In the case there were no compile or runtime errors and the system was behaving as expected the next modifications could be applied otherwise we rolled back to the applied changes and performed bug tracking and bug fixing. The diagram showing the workflow of the porting process can be seen in figure 4.1.

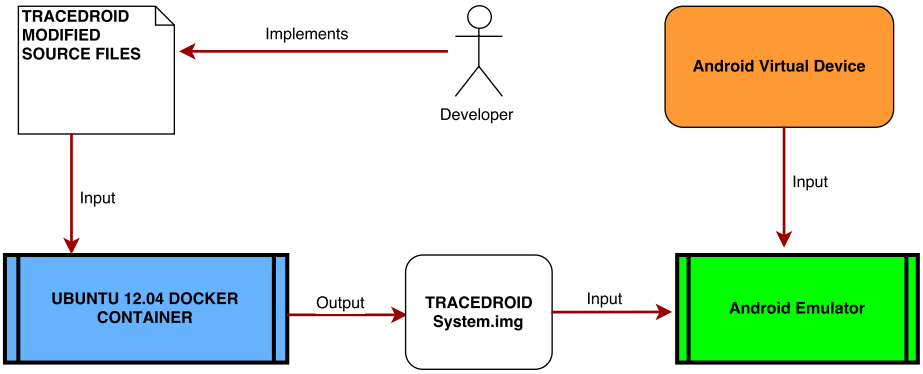

Figure 4.1: Workflow Diagram

4.1.4 Porting steps

We explain in this section the incremental steps taken to perform the porting process. We describe them in a chronological order as follows:

Enabling tracing for a specified app
We added to the file `Globals.h` the variables to deal with the common interface to specify which app to track. The variables hold the uid file containing the uid of the application to be dynamically analyzed. We modified the `Init.cpp` file to implement the functionality of reading the `/sdcard/uid` file and start the tracing in case an app with that unique identifier is launched.

Entering method, exiting method and exception throwing capturing
`dvmMethodTraceAdd()` is the method in `Profile.cpp` responsible for intercepting the calls to enter or exit a method and also the thrown exceptions. Every time this method is called, the action integer-type parameter holds one of the three values, defined by three different macros, specifying what kind of call is caught. In this step we added to `Profile.cpp` the code to analyze this value and, depending on that to call, a method to handle the case. The three methods to work with the three different possibilities were also added, namely `handle_method`, `handle_return` and `handle_throws`.

Getting method indentation and timestamps
In this phase we added the integer-type depth variable to the `Thread.h` header to hold the indentation level. This value is increased before calling `handle_method` and decreased before calling either `handle_return` or `handle_throws`. The `getWhitespace()` method was then ported and the calls to this method added in the three handle ones in order to print correctly the indentation level and the timestamps.

Getting method modifiers, return type and class descriptor
The functions to get the called method modifiers calculated by `getModifiers()`, the return type and the class descriptor, both computed by `convertDescriptor()`, were ported at this stage. The calls to these function were added to `handle_method`.

Retrieving parameters types and values
First the `parameterToString()` method was ported to get a string representation of a parameter containing both its type and its value. Second the `getParameters()` one that iteratively computes the string representations of all the parameters. It accomplishes this by calling the previously described method for each parameter. The array containing the parameters' string representation is then returned. Last the `getParameterString()` function was implemented. It gives a string representation of the array computed by `getParameters()`. The calls to these computational procedures were then added to `handle_method` with an additional check to call them only in the case there are parameters to be computed.

Retrieving exceptions and return value
To compute the string representation of a thrown exception or a returning value we first needed to modify the header declaration (`Profile.h`) regarding `dvmMethodTraceAdd()`. Furthermore, we also redefined the two macros responsible of calling this method in case of exception or returning method, respectively `TRACE_METHOD_UNROLL` and `TRACE_METHOD_EXIT`. The modification was implemented by adding an extra value to hold either the thrown exception or the returning value. The new parameter was also added to `handle_return` and `handle_throws`. The returning value or the exception risen were added in the files were the macros to unroll or exit a method were present. We added, as a last step, in `handle_return` and `handle_throws` respectively `parameterToString()` and `convertDescriptor()` functions to get a string representation of the returning value and the exception.

Printing the traces to `/sdcard/`
First the file pointer to the output file and the lock to perform the writing were added to `Thread.h`. The `prep_log()` function was implemented in `Profile.cpp` in order to create the traces in files that follows the format `dump.ProcessID.ThreadID`. The `ALOGD_TRACE` macro was declared in the `Profile.h` header to actually print the retrieved values to the appropriate dump file. The last step was to add this macro to the three handle methods functions to print the values calculated by the previously ported functions.

4.2 Porting limitations

Features and limitations of our implementation are discussed in this section. Most of the original code concerning TraceDroid based on Android 2.3.4 has been successfully adapted to the sources of Android 4.4. There are, however, some functionalities that could not be ported due to bugs or unexpected behavior. We give in table 4.1 a comprehensive list of the main features of the system, in which file they are implemented and their porting status. The shorthand 'P' for ported, 'NP' for not ported and 'T' for testing is used. A brief description of the features listed follows the table.

Table 4.1: Overview of the ported features status

Feature	File	Status
uid interface	Init.cpp	P
Distinguish method enter, exit and exception throwing	Profile.cpp	P
Tracing return values when profiling a method exit	Profile.h	P
Tracing exceptions when a thrown statement is caught	Profile.h	P
Compute method calls' timestamps and indentation	Profile.cpp	P
Compute method modifiers	Profile.cpp	P
Compute a method's return type and class descriptor	Profile.cpp	P
Compute string representation of parameters and return value	Profile.cpp	P
Filter calls executing bytecode from a system jar	Profile.cpp	T
Compute string representation of a Java object	Profile.cpp	NP
Creating trace files under /sdcard/	Profile.cpp	P
Printing traces to dump files	Profile.h	P

uid interface
Allows to trace a specified app by storing its uid in the `/sdcard/uid` file.

Distinguish method enter, exit and exception throwing
Makes it possible to produce different traces based on whether a method entrance, a method exit or an exception is throw is being profiled. `dvmMethodTraceAdd()` detects the type of action and calls a method to handle it. The three methods that can be called are `handle_method`, `handle_return`, `handle_throws`.

Tracing return values when profiling a method exit
The `TRACE_METHOD_EXIT` macro declaration is augmented to hold the returning value of a method. The value is passed as a parameter from the calls to this macro present in the files under `dalvik/vm/`.

Tracing exceptions when a thrown statement is caught
The `TRACE_METHOD_UNROLL` macro declaration is augmented to hold the thrown exception of a method. The value is passed as a parameter from the Exception.cpp file.

Compute method calls' timestamps and indentation
Implemented by `getWhitespace()` method.

Compute method modifiers
Implemented by `getModifiers()` method.

Compute a method's return type and class descriptor
Both the functionalities are implemented by the `convertDescriptor()` method.

Compute string representation of parameters and return value
The string representation of a returning value is calculated by `parameterToString()`. Since parameters can be more than one, the computation of their string representation is implemented in `getParameters()` and `getParameterString()`. The former returns an array of string representations, the latter a singular string representation with the content of this array.

Filter calls executing bytecode from a system jar
This feature, implemented in the `dvmMethodTraceAdd()` method, allows to omit the traces regarding methods that execute bytecode from a system jar where also the caller consist of bytecode from a system jar. Not tracing these calls speeds up the system and excludes to profile calls that are assured not to contain malicious code. It has been implemented completely differently due to unexpected behavior of the original one that was ported.

Compute string representation of a Java object
This functionality, originally implemented by the `objectToString()` method, consists in getting the string representation of a parameter or returning value if it is a Java object. The method calls the object's Java class `toString()` method in the case there is one. A boolean variable, called `inMethodTraceAdd()`, is also needed in `dvmMethodTraceAdd()` not to trace the calls to the `toString()` method.

Creating trace files under `/sdcard/`
Implemented by `prep_log()` method. It creates dump files for the current process for each of its threads. The process id and the thread id is part of the dump filenames (`dump.PID.TID`). This allows distinguishing between the traces of each thread of the profiled application.

Printing traces to dump files
Implemented by the `ALOGD_TRACE` macro. It acquires a lock to write to the dump files and then it writes the trace logs to the thread dump file.

As it can be seen in table 4.1 most functionalities have been successfully ported and adapted to Android 4.4 source code. The features with the 'T' status have not been included in the stable version of the system since the code implementing them is still in testing. They are included in a separate source code branch with a different compiled system image.

In the stable version of TraceDroid, hence, the filtering of calls executing bytecode from a system jar was not included. This results in larger output traces for the application we are profiling, increasing also the computation steps required by the system. The second feature that has not been ported was about computing the string representation of the parameters and returning Java objects. It has been substituted by printing the class descriptor of the object and the keyword *"Object"* to highlight that the parameter or the returning value is a Java object created by that class.

Chapter 5

Evaluation

In this chapter we evaluate the ported framework for dynamic analysis to understand whether the compatibility rate has been increased from the previous version. We first discuss the testing process and then the analysis of the results.

5.1 Testing

The aim of this phase is to test the implemented system in order to obtain relevant output that could be later analyzed to understand if the porting was successful. The first step was to decide which applications to give as input to the new TraceDroid version, dubbed TraceDroidv4.4. We chose to use the system with a default Android 4.4 application and with 3 external ones not compatible with TraceDroid based on Android 2.3.4. On one hand, testing a default application gave us the output to understand if the system was producing the expected traces, on the other, testing 3 previously-not-compatible applications generated the output to understand if the compatibility of TraceDroid was increased.

The testing environment was the Android emulator with the compiled TraceDroidv4.4 system image and the Android 4.4 AVD as input. We chose the Android browser as the default input application. We stored the app's uid under the common input interface, described in previous chapters, and then we started one of the browser's activity (opening a new blank tab) so that the system could trace its execution. After the applications terminated its execution the trace could be pulled from the `sdcard/` in order to be analyzed.

Aptoide, Firefox and waze were the 3 external applications we decided to use [23, 24, 25]. The following are the reasons behind this choice:

1. All the three applications had a *minSdkVersion* value, in their manifest.xml decompiled with Androguard [12], greater than API level 10, the one used by the previous TraceDroid version, and smaller or equal to API level 19, the one used by TraceDroidv4.4. This is essential to evaluate the compatibility increase of the new system image.
2. They are among the most downloaded Android applications [26, 27].

The environment used for testing and the steps were the same used for the default Android browser except for the need to install the applications since they were not already present in the emulated AVD.

All the 4 used applications were executed with both the TraceDroidv4.4 image producing full traces and the testing one omitting calls executing bytecode from a system Jar. The whole testing process is reproducible and the guides on how it was performed with the resulting output can be found on the project repository [15].

5.2 Result analysis

In listing 5.1 we can see a sample of the default Android browser testing output. As we can see from the output files, the system works as expected: the profile of the app's execution contains all the implemented features (see table 4.2). The output results equal to the original TraceDroid framework with the only, expected, difference of not printing the Java objects because of the limitations of the implementation phase.

Listing 5.1: Default browser sample output

```
1500060766662092:    public void com.android.browser.BrowserSettings$1.run()
1500060766662173:     static android.content.Context com.android.browser.BrowserSettings.access$000
((com.android.browser.BrowserSettings) "Object")
1500060766662258:     return (android.content.Context) "Object"
1500060766662329:     public android.content.res.Resources android.content.ContextWrapper.getResources()
1500060766662402:      public android.content.res.Resources android.app.ContextImpl.getResources()
1500060766662474:      return (android.content.res.Resources) "Object"
1500060766662542:     return (android.content.res.Resources) "Object"
1500060766663040:     public android.util.DisplayMetrics android.content.res.Resources.getDisplayMetrics()
1500060766663123:     return (android.util.DisplayMetrics) "Object"
1500060766663206:     static float com.android.browser.BrowserSettings.access$102(
(com.android.browser.BrowserSettings) "Object", (float) "2513820672.000000")
1500060766663330:     return (float) "1065353216.000000"
1500060766663442:     private static void dalvik.system.VMDebug.startClassPrep()
1500060766663525:      public java.lang.Class java.lang.ClassLoader.loadClass((java.lang.String) "Object")
1500060766663609:       protected java.lang.Class java.lang.ClassLoader.loadClass(
(java.lang.String) "Object", (boolean) "false")
1500060766663704:        final protected java.lang.Class java.lang.ClassLoader.findLoadedClass(
(java.lang.String) "Object")
1500060766663796:         public static java.lang.BootClassLoader java.lang.BootClassLoader.getInstance()
1500060766663870:         return (java.lang.BootClassLoader) "Object"
1500060766663940:         native static java.lang.Class java.lang.VMClassLoader.findLoadedClass(
(java.lang.ClassLoader) "Object", (java.lang.String) "Object")
1500060766664039:         return (java.lang.Class) "Object"
1500060766664107:        return (java.lang.Class) "Object"
1500060766664176:        protected java.lang.Class java.lang.BootClassLoader.loadClass(
(java.lang.String) "Object", (boolean) "true")
1500060766664260:         final protected java.lang.Class java.lang.ClassLoader.findLoadedClass(
(java.lang.String) "Object")
1500060766664341:          public static java.lang.BootClassLoader java.lang.BootClassLoader.getInstance()
1500060766664413:          return (java.lang.BootClassLoader) "Object"
1500060766664482:          native static java.lang.Class java.lang.VMClassLoader.findLoadedClass(
(java.lang.ClassLoader) "Object", (java.lang.String) "Object")
1500060766664577:          return (java.lang.Class) "Object"
1500060766664645:         return (java.lang.Class) "Object"
1500060766664722:        return (java.lang.Class) "Object"
1500060766664788:       return (java.lang.Class) "Object"
1500060766664862:      return (java.lang.Class) "Object"
1500060766664933:     return (void)
```

In listing 5.2 we can see a sample of the waze app testing output. Analyzing the 3 tested applications' output files we can observe that all the TraceDroidv4.4 implemented features (see table 4.1) are working as expected. As for the default browser output, the only difference with the original framework is the absence of the Java objects string representation in the profile of the executed app. These results are fundamental in understanding the app compatibility increase of the ported system.

Listing 5.2: Waze sample output

```
1500218942799907:            new com.waze.NativeManager((null))
1500218942799993:             native public boolean java.lang.Class.desiredAssertionStatus()
1500218942800087:             return (boolean) "false"
1500218942800185:             new java.lang.String((java.lang.String) "Object")
1500218942800274:             return (void)
1500218942800341:             public char[] java.lang.String.toCharArray()
1500218942800428:              native public static void java.lang.System.arraycopy(
(java.lang.Object) "Object", (int) "-1781292680", (java.lang.Object) "Object", (int) "0", (int) "11")
1500218942800549:              return (void)
1500218942800614:             return (char[]) "Object"
1500218942800739:             new com.waze.utils.TicketRoller$Type((null))
1500218942800906:              new com.waze.utils.TicketRoller$Type((java.lang.String) "Object", (int) "0")
1500218942801006:               new java.lang.Enum((java.lang.String) "Object", (int) "0")
1500218942801135:               return (void)
1500218942801201:              return (void)
1500218942801346:              new com.waze.utils.TicketRoller$Type((java.lang.String) "Object", (int) "0")
1500218942801436:               new java.lang.Enum((java.lang.String) "Object", (int) "1")
1500218942801523:               return (void)
1500218942801587:              return (void)
1500218942801674:              new com.waze.utils.TicketRoller$Type((java.lang.String) "Object", (int) "0")
1500218942801761:               new java.lang.Enum((java.lang.String) "Object", (int) "2")
1500218942801844:               return (void)
1500218942801907:              return (void)
1500218942801993:              new com.waze.utils.TicketRoller$Type((java.lang.String) "Object", (int) "0")
1500218942802090:               new java.lang.Enum((java.lang.String) "Object", (int) "3")
1500218942802176:               return (void)
1500218942802240:              return (void)
1500218942802381:              new com.waze.utils.TicketRoller$Type((java.lang.String) "Object", (int) "0")
1500218942802470:               new java.lang.Enum((java.lang.String) "Object", (int) "4")
1500218942802554:               return (void)
1500218942802622:              return (void)
1500218942802839:             return (void)
```

Chapter 6

Conclusion

In this paper we presented an update of the TraceDroid dynamic analysis framework. First we provided an overview of the reasons why the system was originally created and why it is still relevant nowadays. We also provided the necessary background information, in chapter 2, to understand how the system is working. Secondly we explained the TraceDroid design and the porting process implementation, in chapters 3 and 4. Lastly we showed, in chapter 5, that the data obtained in the evaluation phase confirm the success of porting the system from Android 2.3.4 (codename Gingerbread) to Android 4.4 (codename KitKat).

The project repository [15] contains all the technical guides, the compiled images, and the tests output to reproduce the porting process, to use the system and test it. The guides, furthermore, provide information on how to create an Android Virtual Device and emulate the new system image: this should lead to a straightforward integration of TraceDroidv4.4 with TAP.

Android 4.4, the new system version, has still an high market share quota among the OS versions developed by Google but, nevertheless, it will decrease in the next years. The future of TraceDroid will be a complete redesign and reimplementation of the system to adapt it to the new Android runtime (ART) of the most recent versions. Artdroid [28] already shows that it is possible to run a sandbox for dynamic analysis in this runtime. Google provides also, in the source code, some files where to implement the profiling of an application execution [29].

Bibliography

[1] Van Der Veen, Victor, Herbert Bos, and Christian Rossow. *Dynamic analysis of android malware.* Internet & Web Technology Master thesis, VU University Amsterdam (2013).

[2] Unuchek, Roman. *Mobile Malware Evolution 2016 - Securelist.* Securelist.com, 2017, https://securelist.com/mobile-malware-evolution-2016/77681/.

[3] Stark, Harold. *The Google I/O 2017.* Forbes.Com, 2017, https://www.forbes.com/sites/haroldstark/2017/05/20/if-you-couldnt-attend-the-google-io-2017-read-this/.

[4] *Google Play Protect.* Android Open Source Project, 2017, https://www.android.com/play-protect/.

[5] *Millions Of Android Phones Hit By 'Judy' Malware - BBC News.* BBC News, 2017, http://www.bbc.com/news/technology-40092540.

[6] *Android Compatibility Program Overview* Android Open Source Project, 2017, https://source.android.com/compatibility/overview.

[7] *IDC: Smartphone OS Market Share.* Www.Idc.Com, 2017, http://www.idc.com/promo/smartphone-market-share/os.

[8] *App Stores: Number Of Apps In Leading App Stores 2017.* Statista, 2017, https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/.

[9] *Google Play: Number Of Downloads 2010-2016.* Statista, 2017, https://www.statista.com/statistics/281106/number-of-android-app-downloads-from-google-play/.

[10] Neuner, Sebastian, et al. *Enter sandbox: Android sandbox comparison.* arXiv preprint arXiv:1410.7749 (2014).

[11] Unuchek, Roman. *Dvmap: The First Android Malware With Code Injection - Securelist.* Securelist.Com, 2017, https://securelist.com/dvmap-the-first-android-malware-with-code-injection/78648/.

[12] *Androguard.* Github, 2017, https://github.com/androguard/androguard.

[13] Brahler, Stefan. *Analysis Of The Android Architecture.* 2010.

[14] David Ehringer. *The Dalvik Virtual Machine Architecture.* 2010.

[15] *Tracedroid 4.4 Repository.* Github, 2017, `https://github.com/dda410/Bproject`.

[16] *Android Dashboards.* Android Open Source Project, 2017, `https://developer.android.com/about/dashboards/index.html`.

[17] *Android Dashboards.* Android Open Source Project, 2017, `https://developer.android.com/about/dashboards/index.html`.

[18] *Establishing A Build Environment.* Android Open Source Project, 2017, `https://source.android.com/source/initializing`.

[19] *Downloading The Android Source.* Android Open Source Project, 2017, `https://source.android.com/source/downloading`.

[20] *Android Studio.* Android Open Source Project, 2017, `https://developer.android.com/studio/index.html`.

[21] *Command Line Tools Android Studio.* Android Open Source Project, 2017, `https://developer.android.com/studio/command-line/index.html`.

[22] *Android Debug Bridge.* Android Open Source Project, 2017, `https://developer.android.com/studio/command-line/adb.html`.

[23] *Aptoide.* En.Aptoide.com, 2017, `https://en.aptoide.com/`.

[24] *Firefox Android Browser.* Mozilla, 2017, `https://www.mozilla.org/en-US/firefox/android/`.

[25] *Free Community-Based Navigation App.* Waze.Com, 2017, `https://www.waze.com/`.

[26] *List Of Most Downloaded Android Applications.* En.Wikipedia.Org, 2017, `https://en.wikipedia.org/wiki/List_of_most_downloaded_Android_applications`.

[27] *Aptoide Downloads* En.Wikipedia.Org, 2017, `https://en.wikipedia.org/wiki/Aptoide`.

[28] Costamagna, Valerio, and Cong Zheng. *ARTDroid: A Virtual-Method Hooking Framework on Android ART Runtime.* IMPS@ ESSoS. 2016. APA

[29] *Android 7 Trace.* Android Open Source Project, 2017, `https://android.googlesource.com/platform/art/+/master/runtime/trace.cc`.