Alexander Marsalek, BSc

An Android Malware-Detection Framework based on Dynamic Analysis

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Univ.-Prof. Ph.D. Roderick Bloem
Dipl.-Ing. Dr.techn. Peter Teufl
Dipl.-Ing. Daniel Hein

Institute of Applied Information Processing and Communications (IAIK)

Graz, December 2015

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

Abstract

During the past years, smartphones have evolved to omnipresent accessories. For many users, smartphones play a central role in their daily lives. They produce and store sensitive data such as location information, photos, voice recordings, contacts, appointments, and messages. In this regard, especially Android shows a high level of functionality and flexibility. Its openness, extendability, and other features make it the most popular mobile operating system. Android's popularity and the presence of sensitive data have recently attracted the attention of malware authors. The majority of mobile malware samples found in 2013 target Android devices. Thus, sensitive data stored on Android devices must not be assumed to be secure. To address this issue, a malware detection framework for Android is proposed. Based on the proposed framework, two analyses have been developed. The first analysis is based on the DroidBox Application Sandbox. It aims to detect malware that leaks sensitive data or produces costs for the user by sending short messages or starting phone calls. The second analysis is called Tripwire. It aims to detect malware that use root exploits, by analysing file-system changes. Obtained results reveal that several malware families can be reliably detected with the proposed framework. However, dynamic analysis techniques are not suited for all families. Dynamic analysis techniques have the advantage that they are not hampered by encryption, code obfuscation, and dynamic code loading. The drawback is that only one execution path is examined. This means the malicious code is only analysed if it is executed. Despite its limitation, the proposed framework contributes to a reliable automated malware detection on Android devices.

Keywords. Android, Malware, Dynamic Analysis, Framework, Malware Detection, DroidBox, Tripwire, exploits, Monkey

Acknowledgments

First and foremost, I want to thank my advisors Roderick Bloem, Peter Teufl and Daniel Hein for their guidance and their endless support. Their encouragement and feedback helped me during all phases of this thesis.

Besides my supervisors, I also want to thank my colleagues at the Institute of Applied Information Processing and Communications for the stimulating environment.

Finally, I would like to thank my family and my friends for their comprehensive support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Android is the world's most popular mobile platform [1] with over 400 million activated devices [2] and a global market share of 81.3% [3] in the third quarter of 2013. During the Google I/O[1] event held in June 2012, Google announced that over a million of Android devices are activated every day.

Android allows the user to extend the smartphone functionality with applications. Applications are distributed via markets that provide thousands of applications for Android, *Google Play*[2,3] alone, the official market from Google, offers over 600,000 [1] applications and games which are downloaded over 1.5 billion times a month [4].

Android provides an easy interface for developers to access many resources on the smartphone. For example, Android applications can access resources, like the camera, the microphone, the Internet, the location, or the contacts. Android apps can even send short messages or start phone calls. These features allow developers to create powerful applications.

Some of these resources provide access to sensitive data. Consequently, the access to these resources is regulated by security policies. Android offers many security features, such as an application sandbox, Bouncer, and the Android permission system.

Every Android application is executed in a sandbox. If it needs access to a resource

[1] https://developers.google.com/events/io/

[2] https://play:google:com/store

[3] Google Play merges Android Market, Google Music and the Google eBookstore[4]

[4] http://googleblog.blogspot.co.at/2012/03/introducing-google-play-all-your.html

outside of the sandbox, it has to declare the respective permission. At the time of installation, the user sees an overview of all requested permissions and can either allow or abort the installation of the application.
Another security feature of Android is *Bouncer*. *Bouncer* is a cloud-based Android application scanner. Every application that is uploaded to the *Google Play* store is scanned automatically. Moreover, Android supports scanning applications installed from unknown sources[5] with *Bouncer*.
Google improves the security of Android with every version [5], but most devices run with an outdated Android version. This problem is called the fragmentation problem. On December 3, 2012 ended a 14-day period, where Google collected data of Android devices that accessed *Google Play*, with the result that 64.2% of devices run with *Android Gingerbread*[6] or below [6]. Only 0.8% of the devices run with Android Jelly Bean, the then current Android version. Using an old Android version can cause security problems, as these versions may contain security flaws.
The popularity and presence of sensitive data on the devices enticed malware authors [7–13]. In the year 2012 Trend Micro[7] found 25,000 Android malware samples in the wild [13]. Heise estimates that there are about 300 malware families [14]. A malware family consists of samples with similar code and behaviour. At the end of 2013 Kaspersky[8] had already found 148,778 mobile malware samples, belonging to 777 families [15]. The majority of these mobile malware samples (98.05%) targeted Android devices.
Mobile threats can be divided into three types, grayware, personal spyware, and, the most dangerous type, software with malicious intends, also referred to as malware.
Basically, there are two ways how malware can achieve its goals on Android. The first method is to request the necessary permissions, and the second method is to use a root exploit to obtain root privileges, which circumvents the security model of Android. Zhou and Jiang collected and analysed 1260 Android malware samples, belonging to 49 malware families [16]. 36.7% of the samples use root exploits to attain privileged control. 45.3% of the samples make phone calls or send short mes-

[5] Applications not installed by Google Play
[6] Android 2.3.x
[7] http://www.trendmicro.com/
[8] www.kaspersky.com

sages without the users being aware of it. 51.1% of the samples steal private data. Several malware detection techniques exist, the most relevant techniques are static and dynamic program analyses. Other methods to detect malware are for example application permission analysis, cloud-based detection, or battery life monitoring [17]. Several tools are available that are based on these and other methods, each with its advantages and disadvantages.

The goal of this thesis is to create, in joint work with Bergler [18], a fully automatic Android malware-detection framework that is easily usable and extendable. Consequently, the framework needs to be compatible with existing Android tools. To be easily extendable, we decided that the framework should support *Plugins*.

To demonstrate the capabilities of the framework and the *Plugin* system two *Analyses* are created. In this work, the term *Analysis* refers to a framework component that defines an analysis workflow. This workflow mainly defines which *Plugins* should be executed in which order, on which device, and on which applications. Usually, an *Analysis* will produce a result, for example, the two developed *Analyses* classify the analysed applications into suspicious and unsuspicious applications. Both *Analyses* introduced in this thesis are based on dynamic detection techniques. Bergler [18] created some *Analyses* based on static detection techniques. For more details on the static capabilities of the framework refer to [18].

The first dynamic *Analysis* aims to detect malicious applications that steal private data, send short messages or initiate phone calls. An existing tool named DroidBox, which is an Android application sandbox, fulfils all these goals and even more [19, 20]. To demonstrate the powerfulness of the *Plugin* system this tool is integrated as *Plugin*. DroidBox is based on TaintDroid; it combines taint tracking with API hooking to monitor applications. The *DroidBox Analysis* basically starts an emulator running the DroidBox system and subsequently installs and uses all applications. After the execution of an application the monitored events are analysed and a *suspicious score* is calculated. Before the execution of the next application the emulator is restored to a known clean state.

The second *Analysis* aims to detect applications that use root exploits. To achieve this goal the *Tripwire Plugin* was implemented. The *Plugin* is named after the Unix tool Tripwire developed by Kim et. al. [21]. Like the UNIX tool, this *Plugin* is a file

system integrity checker. It compares two snapshots of the file system and analyses the changes. The first snapshot is made before the installation and execution of the application under analysis, the second snapshot is created afterwards. The *Tripwire Analysis* basically creates an initial snapshot of a clean system and subsequently installs and executes all applications under test. After the execution of a suspicious application the file-system changes are analysed and a *suspicious score* is calculated. After this, the device is restored to a clean state. As the operating system is potentially not trustworthy after the execution of potential root malware, the filesystem examination and the restore process are executed in the recovery mode. The recovery mode is a separate minimal operating system that is completely independent of the normal operating system. Both *Analyses* are evaluated against the 1260 malware samples from the Malgenome project [16].

The remaining thesis is structured as follows. The next chapter gives some background by introducing the Android platform with a focus on its security features, the different Android malware types and the malware detection methods. As part of the dynamic detection methods the tools TaintDroid and DroidBox are discussed. Chapter 3 gives an overview of work related to Android malware detection and collection. Chapter 4 introduces the developed framework and its components. Chapter 5 presents the developed *Plugins* and *Analyses*. It starts with the two core *Plugins* *DroidBox* and *Tripwire*; then it explains the other *Plugins* needed for the dynamic analysis of Android applications. Finally, the chapter presents the two *Analyses* named after their main *Plugins*. Chapter 6 presents the detection results. First the chapter presents the merged results, then the individual results of the two *Analyses* are described in more detail. The thesis ends with Chapter 7 where the résumé and an outlook is given.

Chapter 2

Background

This chapter starts with an overview of the Android platform, with a focus on its applications and its security features[1]. Then it describes the different Android malware types, and finally it deals with methods to detect malware. It explains static and dynamic methods, and will introduce the dynamic analysis tool TaintDroid and its enhancement, DroidBox, an application sandbox.

2.1 Android Platform

Android is a Linux-based operating system optimised for mobile devices. Initially, it was developed by Android Inc., and acquired by Google in 2005 [22]. Since that time, the code is maintained and further developed by the Android Open Source Project (AOSP), led by Google. At the moment, Android is the world's most popular mobile platform [1]. Unfortunately, the popularity of Android and the existence of sensitive information on the devices attracts malware authors [23].

2.1.1 Boot Process

After turning on the Boot ROM code is executed. This code is hardwired and can only use the internal RAM. The Boot ROM code initializes the system and then it loads the boot loader. The boot loader initializes the external RAM and loads the main boot loader into it. The primary duties of the main boot loader are

[1]The Android versions *KitKat* and *Lollipop* are not covered in this thesis.

to set-up the file system, the low-level memory protection and to provide network support. After the initialization the boot loader loads either the kernel of the main Android operating system or the kernel of the Android Recovery system. Under normal circumstances, the kernel of the main Android operating system is loaded. The kernel has similar tasks as on a normal personal computer. It will initialize everything that is needed for the system to run. Then it loads the init-process as first user space program. The init-process will parse the init.rc script and start the system processes. One of the system processes is called *Zygote*. *Zygote* starts the *Dalvik virtual machine (VM)*. The *Dalvik VM* is the software that executes applications on Android devices. The first started Java component is the *System server*. It starts all Android services, for instance Bluetooth or the telephony manager. Once all services are started the *BOOT_COMPLETED*[2] intent is fired [24–26]. This intent can be used to auto-start applications.

2.1.2 Recovery Mode

The Android recovery mode is a mini operating system, which has its own kernel and is completely independent of the main Android operating system. It can be used even if the main Android system is broken. It is supposed to help the user if there is an issue with the main OS. Several replacement recovery systems are available for typical Android devices. In this thesis the ClockworkMod-Recovery[3,4] operating system is used. It is a special recovery system that was developed by Koushik Dutta. Compared to the original Android recovery system it provides more features, such as creating and restoring backups, flashing a new ROM, and an *Android Debug Bridge (adb)* shell. For this thesis, the most important feature of the ClockworkMod-Recovery operating system is the ability to create and restore Nandroid backups. Nandroid backups allow to completely backup and restore a complete Android system including all settings, all installed applications and their data.

[2] android.intent.action.BOOT_COMPLETED
[3] http://forum.xda-developers.com/wiki/ClockworkMod_Recovery
[4] http://www.clockworkmod.com/rommanager

2.1.3 Developer Tools

The Android SDK includes several tools that help to develop Android applications. Some of these tools namely *adb*, *logcat*, *Monkey* and *Android Virtual Device (AVD)* are used in this thesis.
The *adb* tool is a command line tool that communicates with an Android device. It supports real devices and emulated devices. The *adb* is a client-server program that consists of three components. These components are a **client** and a **server** which run on the development machine as well as a **daemon** that runs on the Android device. When an *adb* client is started, it first checks whether the **server** is running. If the **server** is not running it is started by the **client**. The **server** connects to all attached Android devices via the **daemon**. The **client** can then execute commands, such as listing all attached devices, installing or removing applications, pulling or pushing files, and opening a shell.
Logcat is a tool that allows to view the log messages of various applications and the system.
Monkey is a program that runs on the Android device and generates a specified number of pseudo-random user and system-level events. Several other options allow to define the behaviour of *Monkey*, for example, how many percent of the events should be of a particular type, or what happens if the application crashes or does not respond. It is commonly used to stress-test applications.
The Android SDK provides an *AVD* manager, which allows to create, modify and delete Android emulators. Several properties can be defined for an *AVD*, such as the Android version, the screen resolution and which sensors are available. Another advantage of the emulator is that it is possible to create snapshots and to revert to them.

2.1.4 Android Applications

Android applications are primarily written in the Java programming language. It is also possible to implement parts in C or C++. The compiled code together with the data, resource files and the **AndroidManifest.xml** are packaged into an Application Package File (APK). The following tree shows the files and folders of a

usual APK:

```
|-- AndroidManifest.xml
|-- classes.dex
|-- resources.arsc
|-- assets
|-- META-INF
|   |-- CERT.RSA
|   |-- CERT.SF
|   `-- MANIFEST.MF
`-- res
```

The **AndroidManifest.xml** defines among others, the name, the version, the necessary permissions and the components of the application. The file itself is encoded in binary XML, meaning it is not human readable. Tools like AXMLPrinter2[5], apktool[6] or Androguard[7] can convert the file into human-readable (plaintext) XML. The **classes.dex** is the Dalvik executable[8], it contains the compiled classes in Java byte code format. The **resources.arsc** file contains compiled resources, such as binary XMLs or layouts. The **assets** folder contains application assets. Android provides special low-level API to open and read these raw files. Android recommends to put raw files like textures or game data into this folder. The **META-INF** directory contains data that ensures the integrity of the APK. The **MANIFEST.MF** file contains the digest of every file of the APK, the **CERT.SF** file contains the digest of the corresponding lines in the **MANIFEST.MF** file. The **CERT.RSA** file contains the signature and the certificate that authenticates the public key that corresponds to the private key used for signing. The name of the **CERT.RSA** and the **CERT.SF** file is variable, only the file extension is fixed [27].

2.1.4.1 Application Components

According to [28] there are four different components that can be used by Android applications. These essential building blocks are **Activities**, **Services**, **Content**

[5]https://code.google.com/p/android4me/downloads/list
[6]https://code.google.com/p/android-apktool/
[7]https://code.google.com/p/androguard/
[8]The Dalvik virtual machine is the software that executes applications on Android devices.

providers and **Broadcast receivers**. The following quotes explain each of these components:

> "*A **Service** is an application component that can perform long-running operations in the background and does not provide a user interface. Another application component can start a service and it will continue to run in the background even if the user switches to another application.*"
>
> Android Developers [29]

> "*A **broadcast receiver** is a component that responds to system-wide broadcast announcements. Many broadcasts originate from the system-for example, a broadcast announcing that the screen has turned off, the battery is low, or a picture was captured. Applications can also initiate broadcasts-for example, to let other applications know that some data has been downloaded to the device and is available for them to use.*"
>
> Android Developers [28]

> "***Content providers** manage access to a structured set of data. They encapsulate the data, and provide mechanisms for defining data security. Content providers are the standard interface that connects data in one process with code running in another process.*"
>
> Android Developers [30]

> "*An **Activity** is an application component that provides a screen with which users can interact in order to do something, such as dial the phone, take a photo, send an email, or view a map.*"
>
> Android Developers [31]

Each component is a different entry point of the application. Three out of the four components are activated by an intent, namely **Activities**, **Services** and **Broadcast receivers**. An intent is a messaging object that can be used to request an

action from a different component [32]. This component can be a part of a different application. For example, if the user clicks on an application icon, the system issues an intent, which starts the application's activity defined in the *AndroidManifest* file. Not all intents are triggered by the user, for example, the system issues an intent if the system is completely booted or if a short message is received. By default, an application is not allowed to receive or issue arbitrary intents. It has to declare the necessary permissions. The Android permission model and other security features are introduced in the next Section.

2.1.5 Security

Android offers many security features, such as sandboxing or the Android permission model [5]. As Android is build on top of the Linux kernel, it uses several Linux security features, such as process isolation, protection of user data, and resource management.

2.1.5.1 Application SandBox

In Android every application runs in a sandbox that isolates the operating system kernel from the applications and the applications from each other. Every Android application gets assigned a unique user identifier (UID) and runs in a separate process, except applications signed with the same certificate. Applications from the same developer, signed with the same certificate have the same UID, because there is no need to protect these applications from each other. Furthermore, the same UID allows easy and secure data sharing using content providers. An application can only access resources outside the sandbox if it has the required permission.

2.1.5.2 Permissions

A limited range of system resources can be accessed by default, but resources that could adversely impact the data on the device, the network, or the user experience are protected by permissions [33, 34]. If an application needs access to critical resources like the location or the contact list, it needs to declare the respective permissions [5]. These permissions are shown to the user at installation time. The

permissions are supposed to help the user to understand what the application is capable of. They can also help the user to recognise suspicious applications. For example, usually a wallpaper application does not need to send short messages. In practice the permission system has a usability problem, the average user is not aware of the impacts of distinct permissions nor of the impacts of the combination of different permissions. Based exclusively on the requested permissions, even experts cannot determine if an application is legitimate or not. For example, an application with access to the Internet and the short message storage is potentially able to leak messages to a third party. However, there is no way for the user to detect this behaviour solely based on the permissions.
Nevertheless even if the user would be aware of the impacts, Android does not allow to deny individual permissions. The user can only accept all permissions or abort the installation process. Another problem is that applications can share their permissions.
That means an application that has the *READ_SMS*[9] permission or the *RECEIVE_SMS*[10] permission and a second application that has the *INTERNET*[11] permission could work together and send the short messages to an adversary's server [35].
Some malware can circumvent the permission system by using root exploits. After gaining root privileges an application effectively gains complete control over the attacked device, the security mechanism are completely broken.

2.1.5.3 Bouncer

Bouncer [36] is the codename of a service developed by Google that automatically scans the Google Play store for malware. Bouncer scans every uploaded application for known malware. Bouncer also executes the applications in Google's cloud infrastructure to look for hidden malicious behaviour.
Oberheide and Miller [37] found several ways to circumvent Bouncer. One way to circumvent Bouncer is to wait five minutes before doing anything bad because Bouncer runs the applications only for five minutes. Another way is to recognise

[9] "android.permission.READ_SMS"
[10] "android.permission.RECEIVE_SMS"
[11] "android.permission.INTERNET"

that the application is executed by Bouncer. This is possible, because Bouncer executes the applications on an emulator and not on a real device. This can be done amongst others by querying the *getprop* attributes or by analysing the output of the *cpuinfo* command.
Android also provides a "Verify apps" option. If it is activated, all installed applications are regularly checked for malware and the device warns or blocks the installation of potentially dangerous applications from other markets and unknown sources.

2.1.5.4 Remote Malware Removal

Android has a feature that allows to remotely remove applications. If a malicious application that poses a threat is detected, it will be removed using this feature. The involved users will receive a notification [38].

2.1.6 Fragmentation Problem

On December 3, 2012 ended a 14-day period, where Google collected data of Android devices that accessed *Google Play*, with the result that 64.2% of devices run with *Gingerbread*[12] or below [6] and only 0.8% of devices run with the, at that time, up-to-date *Jelly Bean*[13] version. As the Android updates bring not only new features, but also security fixes and new security features it may be dangerous to use an outdated Android version [39]. Figure 2.1 shows the distribution.

[12] Android 2.3.x
[13] Android 4.2

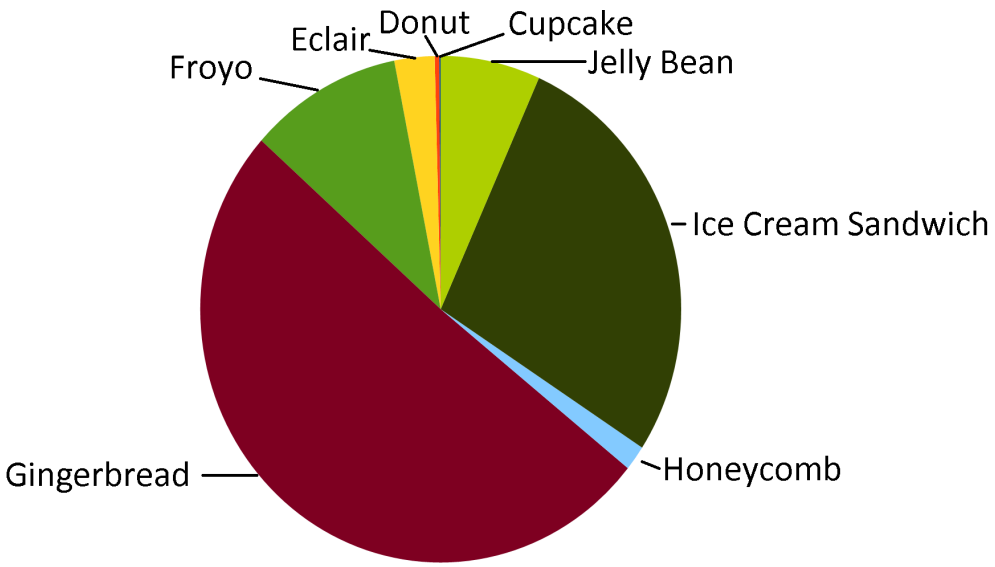

Figure 2.1: Distribution of the different Android versions (Source: [6])[14].

This OS fragmentation is called the fragmentation problem. It is caused by phone vendors and carriers [40]. Delays can be caused because the vendor or carrier typically modify the operating system or add applications. These modifications can not only cause delays; the can also bring additional security threats [41, 42]. Another problem is that keeping old systems up to date is expensive and both, the vendor and the carrier, would rather sell new phones and therefore may decide not provide an update for older phones.

2.2 Malware Types

Unfortunately, the popularity, the presence of sensitive information on the devices, the fragmentation problem and the resultant security problems, make Android an attractive target for malware developers [7–12]. Felt et al. distinguish between three types of mobile threats: malware, personal spyware, and grayware [43]. The distinction is based on the delivery method, legality, and the notice to the user. All three types use different attack vectors and have different motivations. An attack

[14]Portions of this page are reproduced from work created and shared by the Android Open Source Project and used according to terms described in the Creative Commons 2.5 Attribution License.

vector describes how the payload is installed and executed on the device.

2.2.1 Malware

Malware is the most threatening malicious software type. It is build to gain access to a device and damage it, steal data, or violate the data integrity. Malware uses either device vulnerabilities or defrauds the user to get installed. It provides no legal notice and is considered illegal in most countries. Google will remove this kind of application immediately after detection from its store. Zhou and Jiang collected and analysed 1260 Android malware samples that belong to 49 malware families [16]. 36.7% of the samples use root exploits to attain privileged control. 45.3% of the samples make phone calls or send short messages without the user's awareness. 51.1% of the samples steal private data.

2.2.2 Personal Spyware

Personal spyware collects and sends personal information, without the user's knowledge, to the person who installed the spyware. The person who installs the software is probably not the user, but for example his/her spouse. Personal spyware does not send information to the author, therefore its sale is not illegal, but the installation without user's consent may be illegal, depending on the local law.

2.2.3 Grayware

Grayware is not built to harm the user; it has no malicious intention. Grayware typically provides some advantages to the user, but collects user data in the background. The collected data is sent to a remote location, such as an advertisement server, to gain money or to create a user profile. This type of software is normally installed by the user themselves.

2.3 Malware Detection Methods

Chandramohan and Tan review and analyse methods for detecting mobile malware [17]. They differentiate between the following techniques: static analysis, dy-

```
1 SmsManager sm = SmsManager.getDefault();
2 String destinationAddress = "12345678";
3 String text = "Text";
4 sm.sendTextMessage(destinationAddress, null, text, null, null);
```

Listing 2.1: Original Java source code

namic analysis, application permission analysis, cloud-based detection, and battery life monitoring. A short introduction to static and dynamic analysis techniques is given in the following sections. The other three techniques are currently not used by the developed framework and will, therefore, not be discussed in this section. However, a brief description of these techniques is given in Section 3.2.

2.3.1 Static Program Analysis

Static analysis means that the program is analysed without actually executing it. It works on source code or on binaries. Further advantages are that it can detect flaws and threats, and their exact location that cannot be detected by dynamic analysis[15]. While dynamic analysis examines only one execution path, static analysis inspects all paths of a program. If the source code is not available, it is possible to disassemble or decompile the program. There are many tools to disassemble or decompile Android applications. Although decompilers do not reconstruct the original source code, they can achieve good results. For example, Listing 2.2 and Listing 2.3 show the disassembled and decompiled version of the compiled original source code shown in Listing 2.1. For disassembling the tool *baksmali*[16] was used, it produces Smali code. The decompiled code was created using *dex2jar*[17] and *JD-GUI*[18]. Another very powerful tool is *Androguard*[19], it provides many features, such as, disassembling, decompiling, and diffing (calculating the similarities/differences) of Android applications. Static analysis is hampered, if code obfuscation is used [44]. This thesis focuses on dynamic analysis, for more details about static analysis refer to [45–53].

[15] At least not in one run.
[16] http://code.google.com/p/smali/
[17] http://code.google.com/p/dex2jar/
[18] http://java.decompiler.free.fr/?q=jdgui
[19] http://code.google.com/p/androguard/

```
.line 18
invoke-static {}, Landroid/telephony/SmsManager;->
    getDefault()Landroid/telephony/SmsManager;
move-result-object v0
.line 21
.local v0, sm:Landroid/telephony/SmsManager;
const-string v1, "12345678"
.line 22
.local v1, destinationAddress:Ljava/lang/String;
const-string v3, "Text"
.local v3, text:Ljava/lang/String;
move-object v4, v2
move-object v5, v2
.line 23
invoke-virtual/range {v0 .. v5}, Landroid/telephony/SmsManager;->
    sendTextMessage(Ljava/lang/String;Ljava/lang/String;
    Ljava/lang/String;Landroid/app/PendingIntent;
    Landroid/app/PendingIntent;)V
```

Listing 2.2: Smali code (using baksmali)

```
SmsManager.getDefault().
    sendTextMessage("12345678", null, "Text", null, null);
```

Listing 2.3: Decompiled Java byte code (using dex2jar and JD-GUI)

2.3.2 Dynamic Program Analysis

Dynamic analysis means that the application is executed and monitored, mostly in a secure virtual environment, also called a sandbox [54]. For this thesis, there are two relevant techniques:

Behaviour-based analysis monitors the actions performed while the application is executed. One way of monitoring the actions is to hook the Application Programming Interface (API) of the operating system. Hooking is a technique that allows to monitor or alter the behaviour of an operating system by intercepting API calls. For example, if a malware sends a short message using the Android API, this technique is able to abort the sending of the short message or to access or modify the message content and the recipient. The advantage of this method is that it provides very detailed information, such as the arguments and the return value of monitored API functions, the disadvantage is that this technique needs either a modified execution environment or the application itself has to be adapted. DroidBox uses this technique.

Another approach is to use dynamic information flow tracking. Information flow tracking enables to track the information flow between sources and sinks. For this, a taint tag is added to every data retrieved from a specific source. The taint tag is propagated according to a taint policy whenever the tainted data is processed. Let's assume we have the assignment $x = y + z$. If y or z is tainted, then the taint tag will be propagated to x. An advantage of this method is that it can provide very accurate information. For example with this method it is possible to detect that a malware accessed a short message and sent it to a remote server. A disadvantage of this method is that it also needs either an adapted execution environment or a special prepared application. Another disadvantage is that a variety of anti-taint analysis techniques exist. Golam et. al. [55] present some of them against TaintDroid. TraintDroid is an example of a tool that uses this analysis technique.

Difference-based analysis works by comparing two snapshots of the file system and analysing the changes. Usually, one snapshot is created prior the execution of the to be analysed application and one afterwards. Generally, this method is easier to implement and more difficult to detect by malware [20]. However, difference based analysis provides no real time monitoring and not as much information as API hooking. It can only analyse what files have been added, modified, or deleted between two snapshots. For example, this technique is well-suited for detecting permanent root exploits. In this case a binary file could be added by the malware under analysis, which has the SUID (Set User ID) bit set and belongs to the root user. The *Tripwire Plugin* (see 5.1.2) is based on this technique.

2.3.2.1 TaintDroid

This thesis uses DroidBox, which is based on TaintDroid. Therefore, an overview of TaintDroid and DroidBox is given. TaintDroid [56] is a real time information flow tracking system. It uses dynamic taint analysis to track sensitive information through third-party applications.
TaintDroid uses four granularities of taint propagation: variable-level, method-level, message-level, and file-level.

Variable-level tracking within the untrusted application code is achieved by instrumenting the VM interpreter.

Method-level tracking is used by TaintDroid for system-provided native libraries. It means the code is not instrumented, the taint propagation is patched on return. This can be done because these methods have a known information flow. TaintDroid uses a simple heuristic for most methods. This heuristic assigns the union of the methods parameter taint tags to the return value. For selected methods, the authors defined profiles which specify flows between parameters and return values.

Message-level tracking is used instead of tagging data within the message to minimize the Interprocess Communication (IPC) overhead.

File-level tracking is used to ensure that persistently stored information keeps its taint tags.

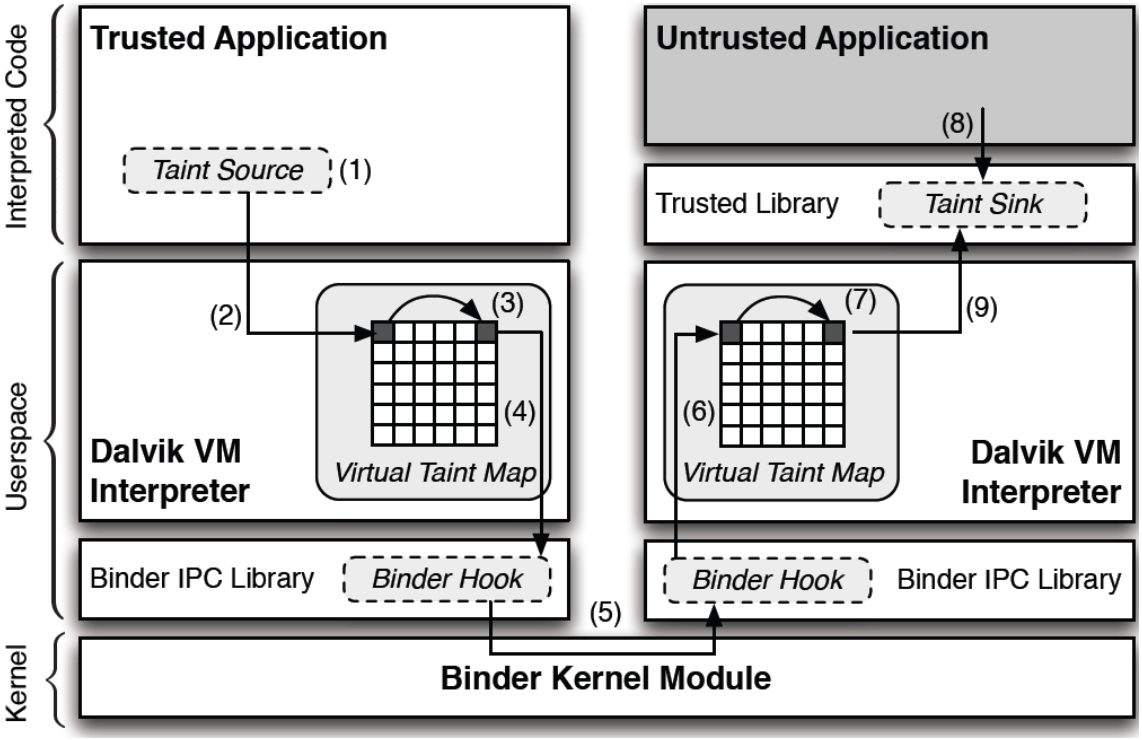

Figure 2.2: Architecture of TaintDroid [Source [56]].

Figure 2.2 shows the TaintDroid architecture. First, sensitive information sources are identified as taint sources (1). For example, the location provider, the camera, and the microphone are used as taint sources. TaintDroid adds taint markings to data retrieved from these sources. The taint information is stored in the virtual taint map (2). A taint propagation logic is used to spread the taint tags, according to data flow rules (for details refer to Table 1: DEX Taint Propagation Logic of [56]) when an application uses the tainted information (3). For example, these rules specify that if a constant value is assigned to a tainted variable, the taint will be deleted or that if a tainted and an untainted variable are added the result will also be tainted. The modified binder library[20] (4) ensures that taint tags are preserved when an application uses IPC transactions. For this, a taint tag is added to the message that contains the combined taint tags of all the data sent. The downside is that this can produce many false positives. Then the message is passed through the kernel (5) and received by the modified binder library of the VM instance of the requesting application. The modified binder library adds taint tags to all values read from the message (6). The taint propagation logic is used to spread the taint tags (7). If an untrusted application uses a taint sink, like the network interface, via a library call, the taint tag is retrieved (9) and the event is reported.

2.3.2.2 DroidBox

DroidBox [19, 20] is an application sandbox that was developed by Patrik Lantz. It allows to dynamically analyse Android applications in an emulator and logs information leaks. For this, DroidBox uses an Android emulator with a modified system image. DroidBox is based on TaintDroid [56] (see 2.3.2.1), but adds more taint sources. The used sources are shown in Table 2.1.
While TaintDroid notifies the user when sensitive information leaves the phone, DroidBox adds a message in JavaScript Object Notation (JSON) format to the Android logging system. The JSON encoding simplifies the processing on the host side. These log messages can be viewed on the PC using the *logcat*[21] tool. Listing 2.4 shows a log message that was created after a sample of the *HippoSMS*[22]

[20]The binder is an IPC mechanism optimised for Android.
[21]http://developer.android.com/tools/help/logcat.html
[22]http://www.csc.ncsu.edu/faculty/jiang/HippoSMS/

Tag	Description	Added
TAINT_ACCELEROMETER	Data from accelerometer	No
TAINT_ACCOUNT	Google account data	No
TAINT_BROWSER	Browser bookmarks	Yes
TAINT_CALENDAR	Calendar data	Yes
TAINT_CALL_LOG	Call history	Yes
TAINT_CAMERA	Data taken by camera	No
TAINT_CLEAR	No taint	No
TAINT_CONTACTS	Contact name and number	Yes
TAINT_DEVICE_SN	Device serial number	No
TAINT_EMAIL	Email data	Yes
TAINT_FILECONTENT	Content of a file	Yes
TAINT_ICCID	ICCID value	No
TAINT_IMEI	IMEI value	No
TAINT_IMSI	IMSI value	No
TAINT_LOCATION	Location	No
TAINT_LOCATION_GPS	GPS coordinates	No
TAINT_LOCATION_LAST	Last known location	No
TAINT_LOCATION_NET	Network location	No
TAINT_MIC	Data recorded by microphone	No
TAINT_OTHERDB	Other database values	Yes
TAINT_PACKAGE	Installed packages	Yes
TAINT_PHONE_NUMBER	Number of the phone	No
TAINT_SETTINGS	System settings	Yes
TAINT_SMS	Numbers and messages	Yes

Table 2.1: This table shows the different Taint sources used by DroidBox [Modified, based on [20]].

```
1 DroidBox: { "SendSMS": { "number": "1066156686", "message": "8" } }
```

Listing 2.4: DroidBox dectects a short message sent by *HippoSMS* malware

malware sent a short message with the text "8" to the number "1066156686".

DroidBox comes with two scripts that are relevant for the user, *startemu.sh* and *droidbox.sh*. First it is necessary to setup an *AVD*. That can be done using the *AVD Manager*. Currently only Android 2.1 and Android 2.3 are supported by DroidBox. Then the *AVD* can be started using the *startemu.sh* script: `./startemu.sh <AVD name>`, where `<AVD name>` is the name of the *AVD*. This

```
1 DroidBox: { "FdAccess": { "path": "2f646174612f646174612f636f6d2e7669727369722↩
    e616e64726f69642e6368696e616d6f62696c6531303038362f73686172265645f70726566732↩
    f73657474696e672e786d6cfc", "id": "1176829910" } }
2 DroidBox: { "FileRW": { "operation": "write", "data": "3c3f786d6c2076657273696f6↩
    e3d27312e302720656e636f64696e673d277574662d3827207374616e64616c6f6e653↩
    d2779657327203f3e0a3c6d61703e0a3c737472696e67206e616d653d2266697273745↩
    f73746172745f74696d65223e312053657020323031322031353a33333a323920474d543c2↩
    f737472696e673e0a3c2f6d61703e0a", "id": "1176829910" } }
```

Listing 2.5: DroidBox detects an app is writing to a file (The path and the data is hex-encoded)

script starts the *AVD* with the modified images. When the emulator is running, the user can start the second script. The second script needs the path of the application that should be analysed as argument and optional the analysis duration: `./droidbox.sh <file.apk> <duration in secs (optional)>`. The script pipes the *logcat* output into a python script. The python script performs a static pre-check and extracts essential information from the *AndroidManifest* file. This information is used to install and start the application using a *monkeyrunner*[23] script. The python script also parses the *logcat* output and generates an analysis report. The analysis report contains the following information:

File hashes: The script calculates the hash value also known as file fingerprint or checksum of the tested application using cryptographic hash functions such as MD5[24], SHA1[25], and SHA256[26].

File activities: DroidBox monitors file activities. The generated information contains the path of the file, the file operation, and the data read or written. Listing 2.5 shows two example log messages. The first message contains the path, and the second message contains the operation and the data written.

Crypto API activities: DroidBox logs the used algorithm, the operation, and the input and output data for cryptographic operations that are performed using the Android API.

[23] http://developer.android.com/tools/help/monkeyrunner_concepts.html
[24] http://tools.ietf.org/html/rfc1321
[25] http://tools.ietf.org/html/rfc3174
[26] http://tools.ietf.org/html/rfc4634

Network activity: DroidBox monitors opened TCP and UDP connections and their incoming and outgoing traffic. For opened connections, the destination address and the port are stored. The source/destination address and the received/sent data is logged for incoming/outgoing network traffic.

DexClassLoader: The path of the loaded DEX-file is logged.

Broadcast receivers: DroidBox also includes the names and actions of broadcast receivers that an application is listening to, in the report. This data is extracted from the *AndroidManifest*[27] file.

Started services: DroidBox logs the names of all started services.

Enforced permissions: DroidBox lists the permissions that are declared in the *AndroidManifest* file in the report.

Permissions bypassed: DroidBox calculates the bypassed permissions from the information that is gathered during runtime and the declared permissions in the *AndroidManifest* file. For example, in earlier Android versions, an application could upload information to a server without the Internet permission by coding the information into a URL and starting an intent which opens the URL in the Android browser [57].

Sent SMS messages: DroidBox logs the destination number and the message content.

Information leaks via file, network and SMS: DroidBox logs if *tainted* data is leaked via file, network, or SMS. For all three types of leaks, the taint tags and the data sent are logged. Additionally, the destination address and port are logged for network leaks. For SMS leaks, the destination number is logged and for file leaks the path and the file operation are logged.

Phone calls: Regarding phone calls, DroidBox logs the destination number.

The time of every event is implicitly contained through the creation time of the corresponding *logcat* entry. As malware could easily detect that it is running in an

[27] http://developer.android.com/guide/topics/manifest/manifest-intro.html

emulator by querying one of the hard-coded values shown in table 2.2, DroidBox uses modified values.

Name	Standard Android 2.1 emulator	Emulator with DroidBox 2.1
ro.product.model	sdk	GT-I9000
ro.product.brand	generic	Samsung
ro.product.name	sdk	Samsung GT-I9000
ro.product.device	generic	GT-I9000
IMEI	000000000000000	
IMSI	3126000000	

Table 2.2: This table shows the values that were changed by DroidBox to hamper emulator detection.

Chapter 3

Related Work

This chapter gives an overview of related work on the topics of this thesis. It starts with a work that discusses the impacts of malware. Then Section 3.2 introduces work related to malware detection broken down to the used analysis techniques. Section 3.3 presents two works related to malware collection. The malware collection *Malgenome* from Zhou and Jiang [16] is used to evaluate the detection rate of the developed framework i.e. its *Analyses*. Section 3.4 presents a work from Park et al. called RGBDroid [58]. RGBDroid does not try to prevent root attacks but responds to them and shields malware from keeping the root privilege. Section 3.5 presents the work of Li, Gu, and Luo [59]. Li, Gu, and Luo propose a systematic Android malware forensic analysis process. Some of the presented tools are used by the framework proposed in this thesis. Finally, Section 3.6 discusses work related to automated testing. As the framework is supposed to analyse Android applications on its own, it needs tools to automatically test or use applications.

3.1 Malware Impacts

Wei et al. discusses which sensitive data is available on Android smartphones, and what impacts malicious and benign applications can have for enterprises [60]. Sensitive data includes the International Mobile Equipment Identity (IMEI), International Mobile Subscriber Identity (IMSI), phone number, contacts, location, SMS messages, and data generated by physical sensors. Malicious applications can

for example, download and install new applications, monitor and exfiltrate data, or call/text premium numbers. However, not only malicious applications are a problem, also insecurely implemented benign applications can cause problems, e.g., through unencrypted storing of account information. Preinstalled applications like HTCLogger (which collected sensitive information, stored it unencrypted and made it available to any application) can cause another problem [42]. The authors present the impacts to the enterprise, for example, loss of privacy, data loss, data integrity loss, monetary loss, and loss of competitive advantage. They also propose some defence strategies, e.g., educating the users, building an enterprise market, using of strict enterprise content management policies, user monitoring, user profiling, and creating of snapshots of the smartphone's software and data.

3.2 Malware Detection

Chandramohan and Tan review and analyse methods for detecting mobile malware [17]. They differentiate between the following techniques: static analysis, dynamic analysis, application permission analysis, cloud-based detection, and battery life monitoring.

3.2.1 Static Analysis

Regarding static analysis, Chandramohan and Tan [17] distinguish between system call based static analysis, static taint analysis, and source code based static analysis. System call based static analysis was proposed by Schmidt et al. [61]. The system calls are extracted and clustered, to classify the application as benign or malicious. Static taint analysis was proposed by Egele et al. [62]. The application is disassembled, and a control flow graph is constructed. Every path from a sensitive source is considered and checked against information leaks, where sensitive information leaves the device. Source code based static analysis is a malware detection method that was proposed by Enck et al. [63]. First the application is decompiled, then a static code analysis suite is used to check the source code.
Another example for static source code analysis is the work from Apvrille and

Strazzere [64]. Apvrille and Strazzere concentrated on finding unknown Android malware (malware that is not detected by any virus scanner). Therefore, they implemented an Android Market scanner which can crawl all existing applications, and not only those, which are visible on a given device. A static analyser checks 39 different properties from 7 categories. These categories are: required permissions, API call detectors, command detectors, presence of executables or zip files, geographic detectors, URL detectors, size of code and special combinations. After the analysis, the static analyser calculates a risk score. If the score is greater than a given threshold the application is marked as suspicious and can be analysed in detail. In contrast, this thesis does not include a Market scanner. Instead the proposed framework is evaluated against the malware samples from the *Android Malware Genome* project [16]. In addition to static analysis the proposed framework also supports dynamic analysis.

3.2.2 Dynamic Analysis

Regarding dynamic analysis, TaintDroid [56] and Android Application Sandbox [54] are introduced. TaintDroid was presented in Section 2.3.2.1. Bläsing et al. [54] created the Android Application Sandbox. It analyses Android applications using static and dynamic analysis techniques. The static analysis decompiles the application and scans for potential malicious patterns, such as, usage of reflection, of the Java Native Interface (JNI), or the `System.getRuntime().exec()` command. The dynamic analysis hijacks systems calls and logs passed information while the application is automatically stimulated by Android *Monkey*[1]. Not all publications relevant for this thesis were covered in [17], therefore, additional work related to dynamic analysis is introduced, starting with DroidBox [19, 20]. DroidBox is an application sandbox and a progression of TaintDroid. It was developed by Patrik Lantz and is very important for this thesis as one of the developed *Plugins* uses the DroidBox images. It was already introduced in detail in Section 2.3.2.2.
The next work was created by Dixon et al., it has similarities to the developed *Tripwire Plugin*. Dixon et al. implemented a PC-based prototype which compares the hashes of all files on the phone, when it is connected to the PC [65]. When the

[1] http://developer.android.com/tools/help/monkey.html

phone is connected to the PC the first time, all hashes are calculated and stored together with the files on the PC. On every subsequent connection, all hashes are calculated again and compared with the stored ones. Only those files that have changed are copied to the PC for scanning. To address rootkits, which could store and send the "correct" hash, the authors are currently exploring mechanisms that look, for example, at the execution time. In contrast to Dixon et al., this work tries to address rootkits by booting into the recovery mode and copying files there. Another difference is that the smartphone is restored to a known state before every analysis. This makes it easier to relate changes to an analysed application.

Next, a community-based approach to detect malware is introduced. Zhao et al. proposed a new framework named RobotDroid to detect malware [66]. It is based on Support Vector Machines (SVMs) using an active learning algorithm. Their approach is to distinguish benign applications from their malicious copies with the same name and version by observing their behaviour with the help of the Android user community.

In [67] Shabtai et al. describe Andromaly. It is an Android malware-detection framework that is light-weight enough to run as a smartphone application. The authors assume that unknown malware can be detected by measuring system metrics and comparing them with system metrics of known malware. The application consists of four main groups: feature extractors, processors, the main service, and the graphical user interface. The feature extractors sample various system metrics like the CPU consumption, the number of network packets sent, the battery level, and the number of running processes. The main service sends that information in form of feature vectors to the processors. The feature vectors are analysed by the processors, which use machine learning techniques. The main service manages the detection flow. The graphical user interface allows the user to configure the application and to explore the collected data. Shabtai et al. had no Android malware available. Therefore, the authors wrote their own malware and tested their framework with them. In contrast to Andromaly, most parts of the proposed framework run on the PC and only some parts run on the smartphone. Furthermore, as malware is available now, the proposed framework is evaluated with real malware.

3.2.3 Application Permission Analysis

Application permission analysis examines the requested permissions of an application. Chandramohan and Tan [17] introduce for example, Kirin, a lightweight mobile phone application certification proposed by Enck et al. [68]. Kirin checks the requested permissions against a set of rules at installation time and alerts the user, if the check fails.

3.2.4 Cloud-Based Detection

Cloud-based detection solves the problem of limited resources (such as CPU power or energy) on a smartphone. They authors introduce Paranoid Android, proposed by Portokalidis et al. [69] and Crowdroid [70].
Crowdroid, developed by Burguera et al. is a behaviour-based malware detection system for Android. The system consists of three components: data acquisition, manipulation of data, and finally data analysis. The first component — data acquisition — is an Android application that collects basic device information, the list of installed applications, and the list of system calls used by the monitored application. Then the collected information is sent to the server, where the other two components process and analyse the data. The second component parses the collected information, stores the basic device information in a database and creates a feature vector. The feature vector contains the call counts of the Android system calls made by the application. The last component uses a k-means algorithm to classify the feature vectors into two clusters to distinguish between benign and malicious applications. The authors of Crowdroid assume that the "good" applications are executed more often. The quality of the analysis will increase with the number of people contributing data by using the application. The application is available in the Google Play Store. The disadvantage of this method is that it only compares the information of different runs of the same application, identified by name and version. While this works well for identifying malware that is added to benign applications, it is not able to detect unknown malware that is not attached to an existing benign application.
Paranoid Android uses a *tracer* to record the necessary information on the smartphone. A proxy mirrors the traffic to the analysis server. The server uses a *replayer*

to replay the recorded information on an emulator. Static and dynamic analysis techniques are used to analyse the sample.

3.2.5 Battery Life Monitoring

Battery life monitoring works by observing the energy consumption on the phone. This approach expects that malicious applications consume more energy than benign applications. Kim et al. [71] and Liu et al. [72] evaluated this approach for Symbian OS.

3.2.6 Summary

Finally, Chandramohan and Tan conclude that static analysis is a fast and cheap technique for detecting malware, but suffers from code obfuscation. Dynamic analysis faces the problems of static analysis, but needs much power and implementation effort.

3.3 Malware Collection

Access to malware is essential to evaluate the recognition rate of malware detection methods. Therefore, two works from Zhou et al. [16, 73], which are related to malware collection, are introduced. The first work describes *DroidRanger*, the second originated the *Android Malware Genome* project. The malware from the *Android Malware Genome* project is used to evaluate the detection rate of the developed framework.

3.3.1 DroidRanger

Zhou et al. collected 204,040 applications from five different markets (Android Market[2] [74], eoeMarket[3], alcatelclub[4], gfan[5], and mmoovv[6]) to analyse the health of these markets [73]. Their software is called DroidRanger and consists of two detection engines, one for detecting known Android malware, and one for detecting unknown Android malware.

The first engine consists of two steps: *permission-based filtering* and *behavioural footprint matching*. The first step *permission-based filtering* aims to quickly reduce the amount of applications, and therefore filters unrelated applications. The authors filter applications by permissions that are essential for malware. For example, the *Zsone* malware family[7] sends SMS messages to certain premium numbers and removes incoming billing-related SMS messages. Therefore, this malware family can be detected by searching for applications that use the *SEND_SMS*[8] and the *RECEIVE_SMS*[9] permissions. The second step *behavioural footprint matching* uses multiple dimensions to describe malicious behaviours, e.g., the used broadcast receivers, or used APIs. To illustrate this, the authors describe a footprint for *Zsone*: Applications that use a receiver for "*android.provider.Telephony.SMS_RECEIVED*" and send SMS messages to specific premium numbers and intercept incoming SMS messages from certain numbers (e.g., by using the *abortBroadcast* method).

The second engine consists of two steps too: *heuristics-based filtering* and *dynamic execution monitoring*. The authors use two heuristics, dynamic loading of new code (using *DexClassLoader*) and dynamic loading of native code from a directory other than the default one (*lib/armeabi*). The *dynamic execution monitoring* records any call related to dynamic loading of Java code and system calls used by existing Android malware (e.g., the *sys_mount* system call). After the execution,

[2] https://play.google.com/store
[3] http://www.eoemarket.com/
[4] http://www.alcatelclub.com
[5] http://www.gfan.com/
[6] http://android.mmoovv.com/web/index.html (not available anymore)
[7] http://blog.mylookout.com/blog/2011/05/11/security-alert-zsone-trojan-found-in-android-market/
[8] "android.permission.SEND_SMS"
[9] "android.permission.RECEIVE_SMS"

the authors analyse the log files and manually validate suspicious applications. The authors detected 211 (32 in the official Google Play store) malicious applications including two zero-day malware samples (one of them in the official Google Play store). The infection rate among the 204,040 applications was 0.02% for the Google Play store and ranged from 0.20% to 0.47% for the alternative markets.

3.3.2 Android Malware Genome Project

Zhou and Jiang accomplished three contributions. First, they present the first large collection of Android malware with 1,260 samples[10] [16]. These samples belong to 49 different malware families. They detected that 86% of the samples were a repackaged version of a benign application, 36.7% contained a root exploit, and 93% of the malware samples had bot-like capabilities. Second, they performed a timeline analysis of the collected malware and characterized them based on their behaviour (e.g., installation type, activation, and payloads). Third, they performed an evolution-based study, which showed that malware is rapidly evolving. The reviewed four anti-virus applications available in the official Google Play store. The result was that the best one (Lookout Security & Antivirus v6.9) detected 79.6% of the samples, while the worst one (Norton Mobile Security Lite v2.5.0.379) detected only 20.2% of the samples. The proposed framework detects about 40% of the samples.

3.4 Malware Defence

Park et al. developed RGBDroid, a Loadable Kernel Module that responds to root attacks [58]. RBGDroid uses a *pWhitelist* and a *Criticallist*. Only processes in the *pWhitelist* are allowed to use root privileges. While malware can still obtain temporary root privileges using a privilege escalation attack, it fails to keep the root privilege. The *Criticallist* protects system layer resources that do not need to be modified, against being manipulated by malware with root privilege. The *Criticallist* contains all resources in the "*/System/framework*" and the "*/System/lib*" directory

[10] http://www.malgenomeproject.org

as well as the "*/System/etc/hosts*" file. Park et al. demonstrated the features of RGBDroid by trying to get a root shell via *adb* and by a *managed code rootkit* attack, and both failed. A *managed code rootkit* attack is an attack that manipulates resources required by a virtual machine (e.g., Framework.jar or Core.jar on Android). The authors measured the performance on an H-AndroSV210 board with Android 2.2. The average I/O throughput was decreased by 7%, and the user program processing time was increased by 7%.

3.5 Malware Forensic

Li, Gu, and Luo propose a systematic process for Android malware forensic analysis [59]. The process consists of three parts: identifying suspicious applications, defeating of the anti-forensic code, and finally recognizing typical malicious behaviour and then deducting of criminal events. Suspicious applications can be identified by their message digest (e.g., build a database for benign applications from the Google Play store), the required permissions (most malicious applications request a list of high-privilege permissions), and the structure of their components (e.g., malware authors often use a receiver for the BOOT_COMPLETED[11] intent). Malware often implements anti-forensics techniques like obfuscation, encryption or environment verification. The proposed countermeasures are decompilation and deobfuscation (e.g., with apktool[12], dex2jar[13] and jd-gui[14]), decryption and program patching (to avoid the environment verification). The authors provide a list of essential functions that are related to malicious behaviour (e.g., cryptographic utilities, self-defined communication protocols, or sensitive data access). Finally, the authors provide a complete forensic analysis of a mobile malware sample from the Honeynet Forensic Challenge 9[15].

[11] "android.intent.action.BOOT_COMPLETED"
[12] http://code.google.com/p/android-apktool/
[13] http://code.google.com/p/dex2jar/
[14] http://java.decompiler.free.fr/?q=jdgui
[15] http://www.honeynet.org/node/751/

3.6 Automated Testing

This section presents work related to automated testing. Kropp and Morales [75] compare the strengths and weaknesses of the Android Instrumentation Framework[16] and the Positron Framework[17] in relation to automated GUI testing. Based on sample code for both frameworks, the authors conclude that the Android Instrumentation Framework provides great flexibility through its low-level API, but needs more test code, which increases the maintenance work and the error rate. The Positron Framework provides a high-level interface which decreases the amount of test code and maintenance work. Compared to GUI desktop testing tools, both frameworks show limitations.

Piotrowski [76] explains some basic test classes for automated testing of Android applications. He explains the Android Instrumentation Framework as well as the Positron framework that is no longer maintained.

The integration of GUI testing frameworks was tested, but none of the tested frameworks allowed a stable and automated analysis of unknown applications. The final version of the developed framework uses the UI/Application Exerciser *Monkey* [77]. The Android *Monkey* tool is a simple tool that runs on the Android device and generates pseudo-random user events.

[16] http://developer.android.com/guide/topics/testing/testing_android.html
[17] http://code.google.com/p/autoandroid/wiki/Positron

Chapter 4

Software Framework

This chapter describes a new novel malware analysis framework developed jointly with Bernd Bergler. The goal is to create an easily usable and extendable framework. Furthermore, the framework should support the automation of analyses in a simple manner. To achieve these goals, the framework is compatible with existing Android developer tools like *adb* or *logcat*. It is designed in a way that allows developers to quickly add new *Analyses*, to rapidly prototype new features and to easily integrate existing tools. The remainder of this chapter discusses the overall architecture of the framework, the execution flow and the interaction between the framework components.

4.1 Architecture

Figure 4.1 gives an overview of the components. The components shown in grey are not required for dynamic *Analyses*. These components play an essential role for static *Analyses* and are described in Bergler's work in full detail [18]. All other components are explained in the following sections.

4.1.1 MDFCore

The *MDFCore* component is the starting point of the framework. It allows to specify which *Analysis* should be executed and which Android applications should be analysed. The framework will then initialize itself and start the defined *Analysis*

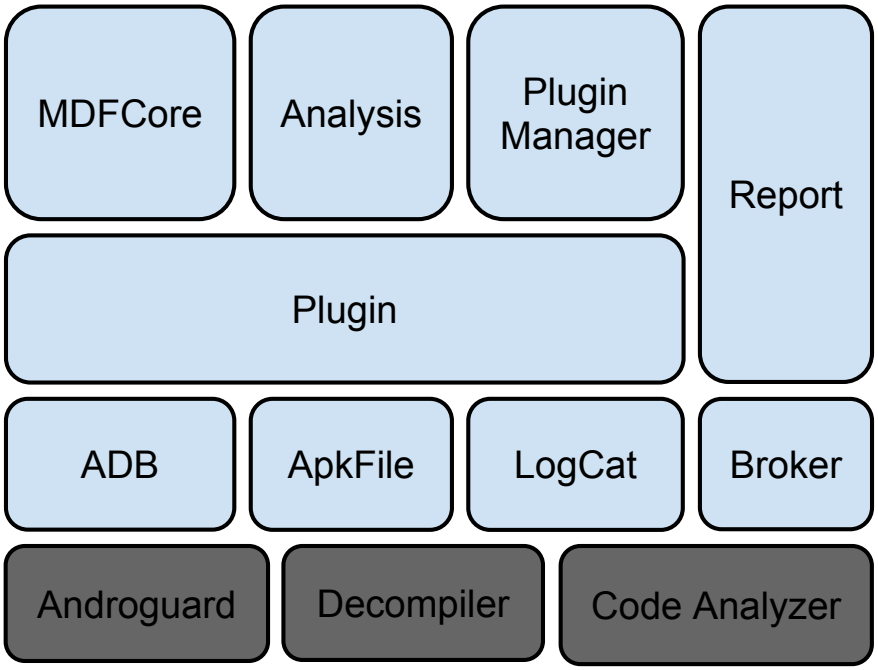

Figure 4.1: This figure shows the components of the framework.

for every specified APK. If for some reason an *Analysis* is interrupted, the framework supports to continue the *Analysis* with the first not analysed application.

4.1.2 *Analysis*

The *Analysis* component defines which *Plugins* should be executed, their order of execution, and assigns, if needed, a device to the *Analysis*. An analysis workflow can use any number of different *Plugins*, but currently it is not supported to use more than one device. Theoretical it would be possible to run several *Analysis* in parallel on different devices, but duo to *adb* instabilities it was decided that only one *Analysis* should run at one time. The *adb* issue will be discussed in Section 4.1.5. The framework supports multiple attached Android devices, which allows to automatically execute an *Analysis* consecutively on different devices. The *Analysis* component has access to the *Plugin Manager*. The developer needs to add the *Plugins* to the *Plugin Manager*. Optionally, the user can pass a configuration to the *Plugin*. The framework will execute the *Plugins* in the order they were added to the *Plugin Manager*. Furthermore, the *Analysis* component has access to the *adbFac-*

tory. The *adbFactory* returns an *adb* component for a given serial number. Every *adb* command will be directed directly to the device with the given serial number.

4.1.3 Plugin Manager

The main task of the *Plugin Manager* is to register and start the *Plugins*. The *Plugin* approach allows to extend the framework's functionality easily. Additionally, the *Plugin Manager* checks if all *Plugin* dependencies are satisfied. *Plugin* dependencies can be specified using annotations and allow to specify that a *Plugin* depends on another *Plugin* that has to be executed before. For example, a *Plugin* that analyses data could depend on a *Plugin* that produces the data to be analysed. A *Plugin* can be configured with an optional configuration object. After the execution of a *Plugin*, the *Plugin Manager* will retrieve its output and add it to the analysis report.

4.1.4 *Plugin*

The *Plugin* component allows to easily extend the framework functionality. If a new tool is integrated into the framework, it should be implemented as *Plugin*. A *Plugin* should address one task, like installing an application. To complete the task, a *Plugin* can use up to four different run phases. The developer can annotate methods to define in which run phase the methods should be executed. The framework supports the following four run phases:

@Init Methods annotated with *@Init* are only executed once per *Analysis*. Methods with this annotation are executed before all other methods. The *@Init* phase is intended for expensive operations that need to be done only once before the *Analysis*, like starting an Android emulator.

@preRun
@Run
@postRun Methods annotated with *@preRun*, *@Run*, or *@postRun* are executed once for every application that should be analysed by the associated *Analysis*. First all methods of all *Plugins* annotated with *@preRun* will be executed, after that methods annotated with *@Run* will be executed and finally all methods annotated with *@postRun* will be executed.

The four run phases allow to map complex processes into one simple *Plugin*. For example, a single *Plugin* can be used to start an emulator with a modified operating system in the *Init* phase, restore the emulator to a known state and start the measurement process in the *preRun* phase and finally analyse the measurement results in the *postRun* phase. Other *Plugins* can use the *Run* phase to start the application under analysis and to use it. The described *Plugin* covers basically the whole DroidBox sequence. It will be described in more detail in Section 5.1.1. Every *Plugin* can use a configuration object and an output object. Both object types can be defined by the *Plugin*. The configuration object is only available, if the framework user registered the *Plugin* together with a configuration object. The output object is always available, its content is added to the analysis report. The output object provides an easy mechanism to permanently store arbitrary data. For details see Section 4.1.9. To fulfil all its tasks, a *Plugin* also has access to the following framework components:

***ApkFile*:** It provides access to the currently analysed APK. For details see Section 4.1.6.

***DataBroker/DataReceiver*:** The *DataBroker* allows to send data to other *Plugins*. If a *Plugin* wants to receive data from another *Plugin* it can use a *DataReceiver*. The *DataReceiver* will be called every time a *Plugin* sends data in a format that the receiver is subscribed to. For details see Section 4.1.8

***AdbWrapper*:** It provides methods for the device interaction, like installing an application. For details see Section 4.1.5.

***Logger*:** The logger allows logging messages with different log levels. These messages will be included into the analysis log file.

4.1.5 AdbWrapper

The *AdbWrapper* component manages interactions with a real device or an emulator. All *adb* commands are directed to the associated device. Therefore, it is no problem if several devices are connected to the PC at the same time. The component provides most of the standard *adb* commands shown at [78] and some other useful features

like loading an emulator snapshot. A description of the most important features of this component follows.

Install: The *AdbWrapper* can install an application on the associated Android device.

Uninstall: The deinstallation of an application is also supported. The package name of the application is required to uniquely identify the application.

Load A SnapShot: This command loads a previously created snapshot, it works only on *AVDs*.

Boot device into recovery: This command reboots the device into the recovery mode and waits until the device state is "*recovery*". This command works only on physical devices.

Monkey: The *AdbWrapper* can also start the Android *Monkey* tool [77]. The user can specify how many pseudo-random events should be sent to the application. Additional the maximal run time of the *Monkey* command can be defined. After this time, the *Monkey* process gets killed.

Copying of data: The *AdbWrapper* supports to copy data from and to an Android device.

Restart Server: This command restarts the *adb* server.

Send Broadcast: This command sends a broadcast. The list of broadcast receivers can be retrieved via the *ApkFile* component.

Start Activity or Service: The *AdbWrapper* component also provides methods to start an activity or a service. The list of activities and services can be retrieved via the *ApkFile* component.

From time to time, the *adb* connection gets lost. This is not a framework issue, but an *adb* problem. If the connection to the device gets lost, the framework tries to re-establish it by restarting the *adb* server. If the framework is not able to re-establish the connection it will inform the user about the problem and will wait

until the connections is restored by manual intervention, such as unplugging and replugging the device. The problem is that while the framework tries to reconnect to a device all other *adb* connections are also closed. Currently, the framework has no mechanism implemented that manages this problem. This is the reason, why the current version does not support the execution of several *Analyses* on different devices in parallel.

4.1.6 ApkFile

The *ApkFile* component provides easy access to the APK itself and to the information stored in the *AndroidManifest*[1] via the *ManifestParser*. This component provides identifiers of the application under analysis, like the MD5 or SHA1 fingerprint and the package name. The package name uniquely identifies an application on the device or the Google Play store. However, it does not uniquely identify a file, because different versions of an application typically use the same package name. Furthermore, the list of activities, services and content providers of the application can be retrieved. It is also possible to retrieve only activities that serve as main entry point of the application[2]. All the information, except fingerprints, is extracted from the *AndroidManifest*. Dynamically generated services or activities cannot be detected using this method, but these services or activities are anyway not accessible from outside the application.

4.1.7 Logcat

Similar to the *logcat*[3] command, this framework component is intended to view messages from the Android logging system. The *logcat* related methods can be accessed through the *AdbWrapper* component. *Plugins* can subscribe themselves to a *logcat* observer to receive the log messages as they come in, or retrieve all collected log messages at once. If the *adb* connection gets lost, the recording of log messages stops. At the moment, this component does not automatically resume the recording when the device is ready again. This means, for example that the developer has to

[1] http://developer.android.com/guide/topics/manifest/manifest-intro.html
[2] http://developer.android.com/training/basics/activity-lifecycle/starting.html
[3] http://developer.android.com/tools/help/logcat.html

start the logging process again after a device reboot or similar events that cause a lost connection.

4.1.8 Broker

The data broker provides an easy way to exchange data between *Plugins*. If a *Plugin* wants to send data, it can pass the data to the broker. Every *Plugin* that implements a *DataReceiver* for the published data type will receive the data. A *Plugin* can publish different data several times, but the receiving *Plugins* have to store the information themselves if they need it, the framework does not store this information for later processing.

4.1.9 Report

The framework creates a report for every analysed application. The report contains all relevant information, like the configuration, the execution time and the output of every *Plugin*. The runtime is stated for all run phases separately. In addition, the report contains the filename of the analysed application as well as a *Boolean* value that indicates if the *Analysis* succeeded or failed. An *Analysis* fails if a *Plugin* throws an exception that is not caught inside the *Plugin*. Every *Plugin* can add arbitrary output data to the report. The report will be stored on the hard disk in XML format. The XML format was chosen because it allows easy, automated processing, and it is human readable.

4.2 Framework Execution Sequence

Figure 4.2 shows the basic execution sequence of an *Analysis* with three *Plugins* and a device. Plugin 1 uses the *Init*, *preRun* and *Run* phase. Plugin 2 uses only the *Init* and the *Run* phase, while Plugin 3 uses only the *postRun* phase. This sequence is intended to help the reader to better understand the different run phases of the *Plugins* and the execution sequence of the framework.
First the framework is started by creating an *MDFCore* object. Then the user defines which *Analysis* should be executed and what applications should be analysed. The

framework executes the *Analysis* for every application that should be analysed. The *Analysis* defines which *Plugins* should be executed and the *Plugin* execution order. In this example Plugin 1 is added (and therefore executed) first, followed by Plugin 2 and Plugin 3.

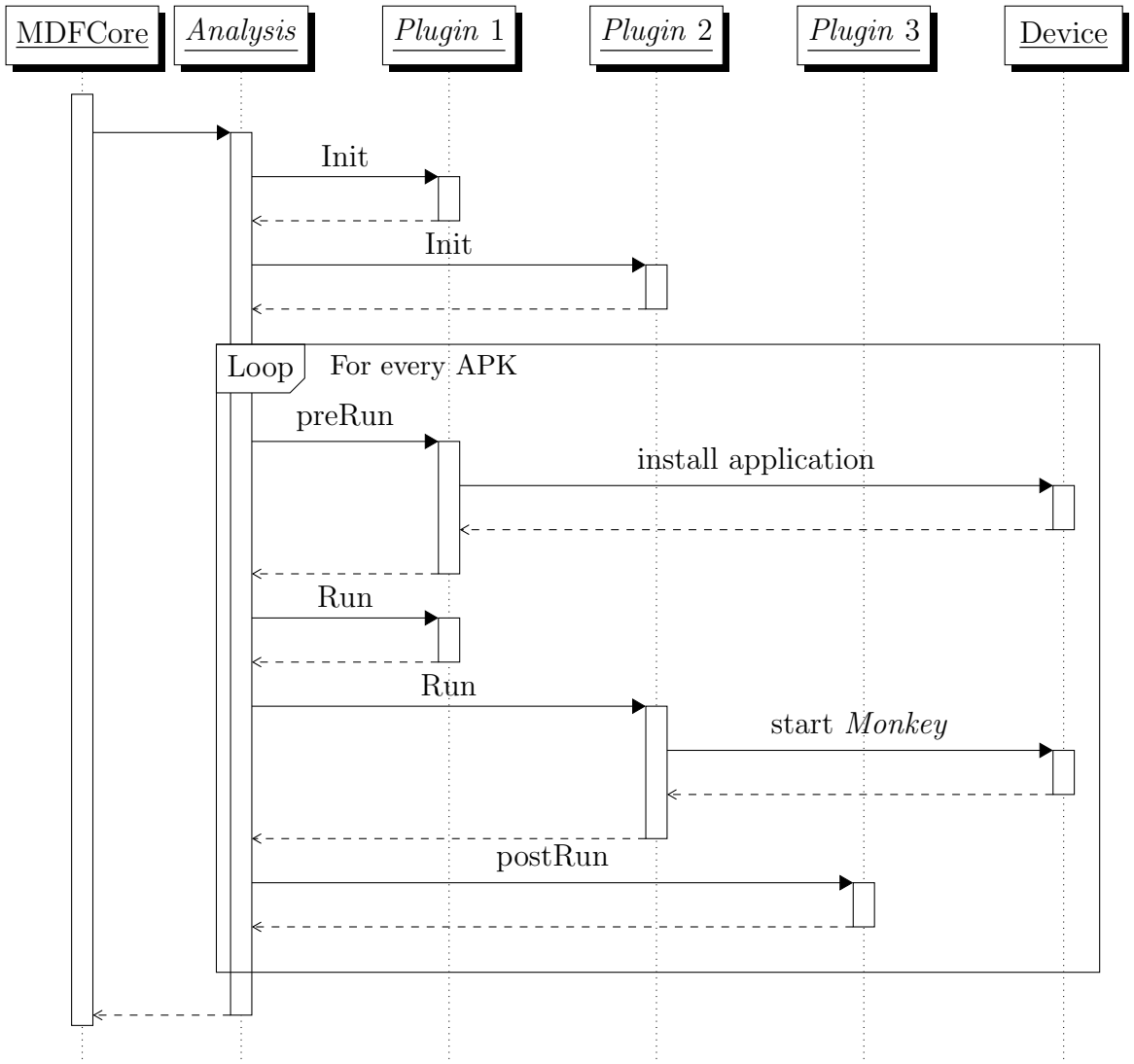

Figure 4.2: An example framework sequence of an *Analysis* with three *Plugins*.

The framework fulfils all its design goals. It is easy to use and it can automatically analyse huge amounts of applications. Furthermore, the *Plugin* component enables developers to easily extend its functionality.

Chapter 5

Plugins and *Analyses*

This chapter presents the developed *Plugins* and *Analyses*. First the *Plugins* are explained with a focus on the *DroidBox Plugin* and the *Tripwire Plugin*. Later on the two *Analyses*, the *DroidBox Analysis* and the *Tripwire Analysis*, are described. These *Analyses* are mainly based on the same named *Plugins*.

5.1 Malware Detection *Plugins*

This section describes the *Plugins*, developed to detect malware. The two main *Plugins*, *DroidBox*, an application sandbox, and *Tripwire*, a file system integrity checker, are described in Section 5.1.1 and 5.1.2 respectively. Section 5.1.3 gives an overview of other *Plugins* needed during the *Analyses*.

5.1.1 *DroidBox Plugin*

The goal of the *DroidBox Plugin* is to integrate the existing Android application sandbox tool named DroidBox into the framework and to extend its functionality. The DroidBox tool and its capabilities were previously introduced in Section 2.3.2.2. To use the tools functionality, the *Plugin* has to start an Android emulator with the DroidBox images and record its logged events. In contrast to the original DroidBox tool, this *Plugin* also analyses the recorded events and calculates a *suspicious score* for every analysed application. Furthermore, the Android emulator is reset to a known state for every application that is analysed. This *Plugin* is an integral part

of the *DroidBox Analysis*, described in Section 5.2. A description of the different run-phases of this *Plugin* follows, a complete *Analysis* sequence that includes this *Plugin* is shown in Figure 5.1 located in Section 5.2. The sequence will be described in Section 5.2.

Init: In the *Init* phase, the *Plugin* starts the emulator with the modified images.

PreRun: In the *preRun* phase, the *Plugin* loads the previously created snapshot, and starts logging the *logcat* output.

Run: The *Run* phase is not used by this *Plugin*.

PostRun: In the *postRun* phase, the *Plugin* stops logging the *logcat* output. Then the *Plugin* filters and parses the *logcat* output into Java classes. Finally, the output is analysed and a *suspicious score* is calculated, according to table 5.1

The values in Table 5.1 were chosen empirically to meet the design goals of this *Plugin* and the *DroidBox Analysis*. One of the goals is to detect malware that sends short messages or starts phone calls, for this the respective DroidBox events "phonecall" and "sendsms" increase the score by ten. For this *Analysis*, a *suspicious score* greater than or equal to ten means that this application is suspicious and should be analysed in more detail. As the DexClassLoader is often used by malware to dynamically load and execute code, the respective event also increases the *suspicious score* by ten. Other suspicious events include the accessing of system files or the storage of sensitive information like the phone number or a short message on the file system. For these actions, the *suspicious score* is also increased by ten. Another goal of the *DroidBox Analysis* is to detect malware that leaks sensitive information. For this purpose, the *DroidBox Analysis* examines all events that involve tainted data leaving the device. Depending on the tainted data that is leaked the *suspicious score* is increased. The values were adapted several times to maximize the detection rate and to minimize the false positive rate. In the end leaked short messages are always considered suspicious, but other tainted data like the location or the IMSI alone are not enough to mark an application as suspicious. Most malicious applications that leak sensitive data steal almost everything they can get. These applications will be detected because this leads to a big accumulated *suspicious score*.

Score	Event		Description
10	DexClassLoader		If the usage of the *DexClassLoader* is detected, the *suspicious score* is increased by ten.
10	fdaccess		If the application accesses a file in the */system* directory, or if the application stores the phone number, the SMS number, or the SMS message on the file system, the *suspicious score* is increased by ten.
10	phonecall		Every phone call made by the application increases the *suspicious score* by ten.
10	sendsms / DataLeakSMS		For every SMS message send by the application the *suspicious score* is increased by ten.
10	TAINT_SMS	DataLeakNetwork	If data leaks via the network, the score is increased depending on the taint tag(s) of the data.
3	TAINT_MIC		
3	TAINT_IMSI		
3	TAINT_IMEI		
3	TAINT_ICCID		
2	TAINT_PHONE_NUMBER		
2	TAINT_LOCATION_NET		
2	TAINT_LOCATION_LAST		
2	TAINT_LOCATION_GPS		
2	TAINT_LOCATION		
2	TAINT_DEVICE_SN		
2	TAINT_CALL_LOG		
2	TAINT_ACCOUNT		
1	TAINT_SETTINGS		
1	TAINT_PACKAGE		
1	TAINT_OTHERDB		
1	TAINT_FILECONTENT		
1	TAINT_EMAIL		
1	TAINT_CONTACTS		
1	TAINT_CAMERA		
1	TAINT_CALENDAR		
1	TAINT_BROWSER		
1	TAINT_ACCELEROMETER		

Table 5.1: This table shows the *suspicious scores* of different events.

5.1.2 *Tripwire Plugin*

This section describes the *Tripwire Plugin*. It was developed to detect applications that gain root access e.g. through an exploit and make changes on the file system that cannot be done without root access. For this, Tripwire compares the file system contents before and after the execution of the application under analysis.
All file information is collected on the Android device in the recovery mode, which works independently from the normal Android operating system. Tripwire compares the following file information to detect changes: the modification date, the file size, the file permissions, the file owner and group, and the file fingerprint. A fingerprinting algorithm maps data of arbitrary size to a fixed length bit string called fingerprint. This *Plugin* uses SHA-1 as fingerprinting algorithm. The file's fingerprint is necessary to reliable detect modifications on the file's content. Such modifications are possible without changing other file information [21]. Without the fingerprint, it would be necessary to copy all files every time from the smartphone to the PC to compare them. The functionality of Tripwire is split into three phases.

Init: In this phase, the *Plugin* restores the connected Android smartphone to a known clean state. Then the *Plugin* collects all important file information and copies all files to the PC. These operations are only executed if the smartphone is used with this *Plugin* the first time. All these steps are executed in the ClockworkMod Recovery mode introduced in Section 2.1.2. For collecting the files and their information the *AdbWrapper* is used. The ClockworkMod Recovery mode is used, because it offers a more powerful shell than the Android operating system and it provides a clean operating system.

PreRun: In this phase, the *Plugin* loads the previously user created snapshot to restore the connected Android smartphone to a known clean state. Starting from a known clean state makes it easier to detect changes and assign them to the analysed application.

Run: The *Run* phase is not used by this *Plugin*.

PostRun: In the *postRun* phase, the *Plugin* detects all file-system changes made between the execution of the *preRun* phase and this phase. For this, the *Plugin*

reboots the smartphone into the ClockworkMod Recovery mode and collects all relevant file information. The ClockworkMod Recovery mode is used, because the Android operating is not trustworthy after the execution of potential root malware. Then it compares the actual information with the previously collected information to identify added, changed, and deleted files. If desired, the *Plugin* copies all added and changed files to the PC for further manual analysis.

These changes are stored and analysed using two methods. The first method checks the fingerprints of the files against a blacklist. The second method searches for suspicious files attributes. At the moment a simple, but effective approach is used. Added files are marked as suspicious if they are stored in the "*/data/data/*" folder and do not belong to an owner and group starting with "*app_*"[1]. The idea behind this rule is that Android creates for every installed application a folder under "*/data/data/app-package-name*" which belongs to the app itself. It is very suspicious, if a file owned by e.g. root is found in this folder. After a few tests, it was clear that this rule also leads to false positives. Some special files like the telephony database[2] or the Bluetooth-settings file[3] belong to others users, like "radio" or "system". The rule was then extended to check the owner, group and permissions for this special files against their default values.

Additional it is suspicious if a file is added, modified or deleted in one of the following directories: */system*, */sbin*, */etc*, or */ref*. These directories contain system files, which are only changed during a system update. Root permissions are required to modify these files. Every suspicious change increases the *suspicious score* of the application by one. The higher this score is, the more likely the analysed application is malicious. Currently, every application that has a *suspicious score* greater than zero is reported as suspicious.

This *Plugin* allows to identify applications that use permanent root exploits or make file-system changes that need elevated privileges. It is not suited to detect

[1] Files in the "lib" folder of the applications are ignored, because they belong not to the application itself.

[2] /data/data/com.android.providers.telephony/databases/telephony.db

[3] /data/data/com.android.settings/shared_prefs/bluetooth_settings.xml

applications that gain temporary root rights and use them in a way that does not leave suspicious file information. For example, a malicious application could exploit the phone, steal sensitive data from the system or other applications and send them to a remote server without being detected by this *Plugin*.

5.1.3 Helper Plugins

This section describes all other *Plugins* used in the *Analyses*. If not stated otherwise the *Plugins* use only the *RunPhase*.

Install Application: The *Install Application Plugin* installs the application in question on the device that is associated with the *Analysis*.

Launch Application: This *Plugin* starts the first launcher activity of a previously installed application. In other words, this plugins simulates an user that clicks on the application's icon to start it.

Kill Application: This *Plugin* kills the application under analysis using the Linux 'kill' command.

Monkey: This *Plugin* is intended to simulate a user. For this, the *Plugin* first starts *Monkey* (cf. Section 2.1.3) limited to the package of the analysed application. *Monkey* will stop after 3000 events or ten minutes. Then this *Plugin* starts all activities and instruments each of them with *Monkey*. This time *Monkey* is limited to 300 events and 3 minutes. If the tested application dies, *Monkey* gets killed.

Simulate Phone Calls and SMS messages: This *Plugin* tries to trigger broadcasts related to short messages and phone calls. For this, the Plugin simulates an incoming phone call that is denied after two to seven seconds. Then the *Plugin* simulates a phone call that is missed. After the missed phone call an incoming short message is simulated, followed by an outgoing phone call and an outgoing short message. On a real device, only the outgoing phone call and the outgoing short message will work.

Start Services: This *Plugin* starts all in the manifest declared services by invoking the *start service* command offered by the *AdbWrapper* component. Starting services will not work with Android 2.1 and prior. This command will fail if the service needs any special parameters.

Send Broadcasts: This *Plugin* sends all broadcasts, that the application is listening to. This command does not specify any options. For example, a "SMS_RECEIVED"[4] broadcast will fail because this *Plugin* does not add the necessary information like the message content or the number. However, the *Plugin Simulate Phone Calls and SMS messages* solves this problem for the most important broadcasts.

Capture Network Traffic: This *Plugin* captures the network traffic. Currently, it works only on an emulator, but it could be easily extended to work on rooted phones. This *Plugin* starts the capturing process in the *preRun* phase and stops it in the *postRun* phase. Currently, the traffic captured by this *Plugin* is not analysed. Instead, the traffic is obtained and analysed using DroidBox. This *Plugin* has the advantage that it creates network captures in the 'libpcap' format which can be analysed with a variety of programs.

5.2 *DroidBox Analysis*

The *DroidBox Analysis* was developed to detect applications that leak sensitive information, use services that cost money, or use suspicious functions as *dexClassLoader*[5]. The main component of the *Analysis* is the *DroidBox Plugin*, which uses the DroidBox system image. For details about the *DroidBox Plugin* see 5.1.1, for details on DroidBox see Section 2.3.2.2 or [19, 20].

Figure 5.1 shows a high-level view of the *Analysis*. First the user has to create an Android emulator, running either Android 2.1 or 2.3 (1). At the time of writing, DroidBox supports only those two versions. The next step is to start the *AVD* with the associated DroidBox images and populate it with data and configure it

[4] "android.provider.Telephony.SMS_RECEIVED"

[5] The *dexClassLoader* is often used by malware to dynamically load a malicious APK that was previously downloaded via the network [79].

as desired (2). The emulator should look like a normal phone to the malware. An empty contact list, no calls in the call log or no received or sent messages could look suspicious to malware, which may prevent the execution of its malicious payload. Two numbers were added to the phone's contact list, and both numbers were called, so that they appear in the call log. Furthermore, a short message was sent to one of the two numbers. Once the desired state is achieved, the user has to create a snapshot (3). This snapshot will be loaded prior to every *Analysis*. Now the emulator is ready, to be used with the automated framework. Steps 5 to 15 are repeated for every application. If the emulator is not running, the *DroidBox Plugin* (see 5.1.1) starts it (4). Then the previously created snapshot is loaded (5). This ensures that every application has the same initial conditions. After the snapshot is loaded, the logging process is started. Then the *Network Capturing Plugin* starts the network capturing process (6). Afterwards, the *Install application Plugin* (see 5.1.3) installs the application to be analysed (7). Then the application is started (8) and killed after fifteen seconds (9). The starting is required, to see what events happen during the start of the application without user interaction. The time of fifteen seconds was chosen as a trade-off between allowing slower applications to start and not to waste too much time. After the fifteen seconds, the application gets killed, so that it cannot disrupt the next *Plugin*. The *Simulate Phone Calls and SMS messages Plugin* (see 5.1.3) simulates in- and outgoing Short Message Service (SMS) messages and phone calls (10). Next the *Monkey Plugin* (see 5.1.3) is started. As first action, *Monkey* tests the main launcher activity, after that *Monkey* is started for every activity declared in the manifest (11). The next two *Plugins*, *Send broadcasts* and *Start services*, are used to trigger as many malicious events as possible. For this, the *Plugin Send broadcasts* (see 5.1.3) issues all broadcasts the application is listening to (12). After that the *Plugin Start services* (see 5.1.3) starts all declared services (13). Then the *DroidBox Plugin* parses and analyses the *logcat*[6] output (14). All relevant information is stored and a *suspicious factor* is calculated. The last *Plugin Network Capturing* stops the network capturing process[7]. The *Analysis* is executed again starting with step 5 for every application in the *Analysis* queue. Steps 7 to

[6]http://developer.android.com/tools/help/logcat.html

[7]The network data captured by this *Plugin* is not used by this analysis, it was recorded for a different project.

13 are done to trigger as much behaviour of the application as possible. The results of this *Analysis* are presented in Section 6.2.

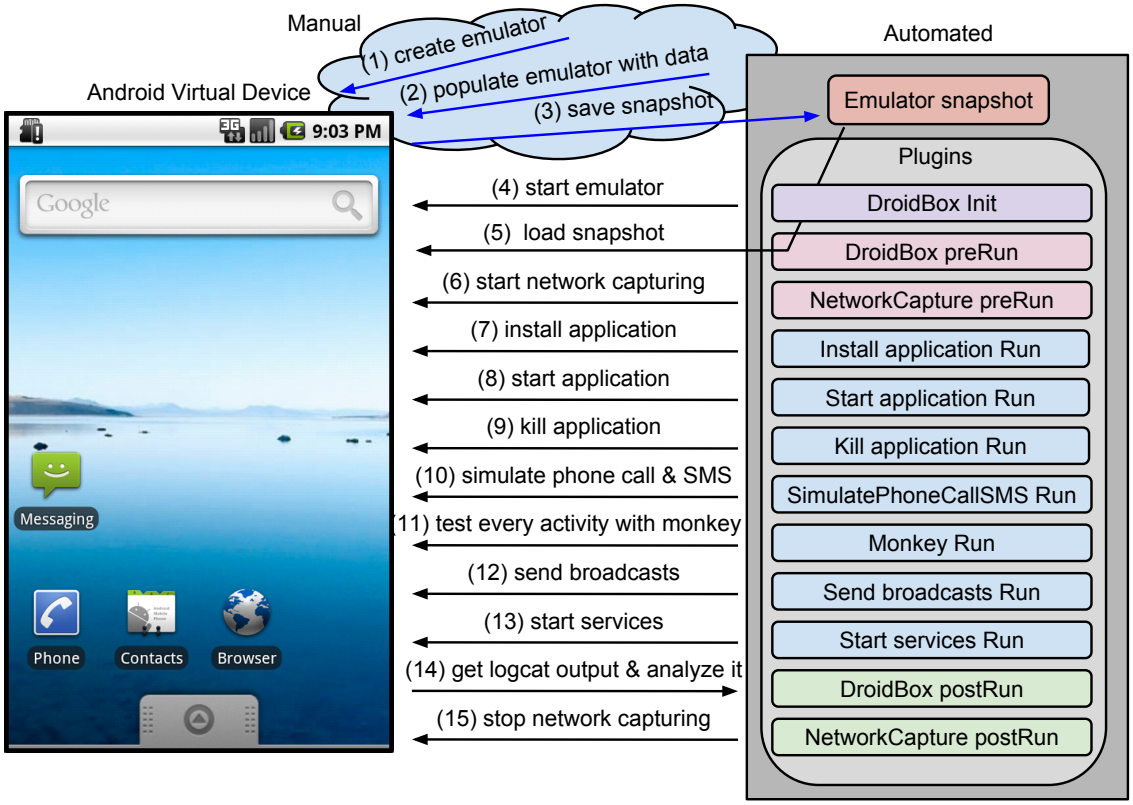

Figure 5.1: The analysis sequence of the *DroidBox Analysis*.

5.3 *Tripwire Analysis*

The *Tripwire Analysis* was developed to detect applications that use permanent root exploits or modify the file system in a way that is not possible without root privileges. This *Analysis* requires a physical device, because the Android emulator has no recovery mode. A Nexus One with Android 2.2 was used for this *Analysis*. First of all "USB debugging" was enabled and the installation of applications from "Unknown sources" was granted. Then the Nexus One, was unlocked and rooted, to be able to flash the ClockworkMod Recovery[8,9] mode. The application "No

[8]http://forum.xda-developers.com/wiki/ClockworkMod_Recovery
[9]http://www.clockworkmod.com/rommanager

Lock"[10] was installed to prevent the screen from locking (1). First tests were made without this application. After every reboot the screen was locked, which was a major problem for most *Plugins*. It is possible to unlock the device using special key-events, but the unlocking did not work reliably enough. The installation of the "No Lock" application solved the problem. Two numbers were added to the phone's contact list, and both numbers were called, so that they appear in the call log. Furthermore, a short message was sent to one of the two numbers (2). After these steps, a backup was created using the ClockworkMod Recovery mode (3). This backup will be loaded prior to every *Analysis*. After these preparations, the device can be used with the proposed malware-detection framework. Figure 5.2 shows a high-level view of the *Tripwire Analysis*. The *Plugins* used by the *Tripwire Analysis* are mostly the same as the *Plugins* used by the *DroidBox Analysis*. In the *Init* phase of the *Tripwire Plugin*, the device is initialized. That means all important information is collected and stored on the PC (4). These steps are only done once for every distinct device backup. After the *Init* phase is finished, the *preRun* phase is executed. In this phase, the *Tripwire Plugin* restores the device with the previously created backup (5). The steps 6 to 12 are equal to the steps 7 to 13 described in the *DroidBox Analysis* (see Section 5.2). The last step in the *Tripwire Analysis* is to collect all important information again, compare it with the previously collected information and analyse the detected changes of the file system (13). Steps 5 to 13 are repeated for every application under analysis.

[10] `https://play.google.com/store/apps/details?id=org.jraf.android.nolock`

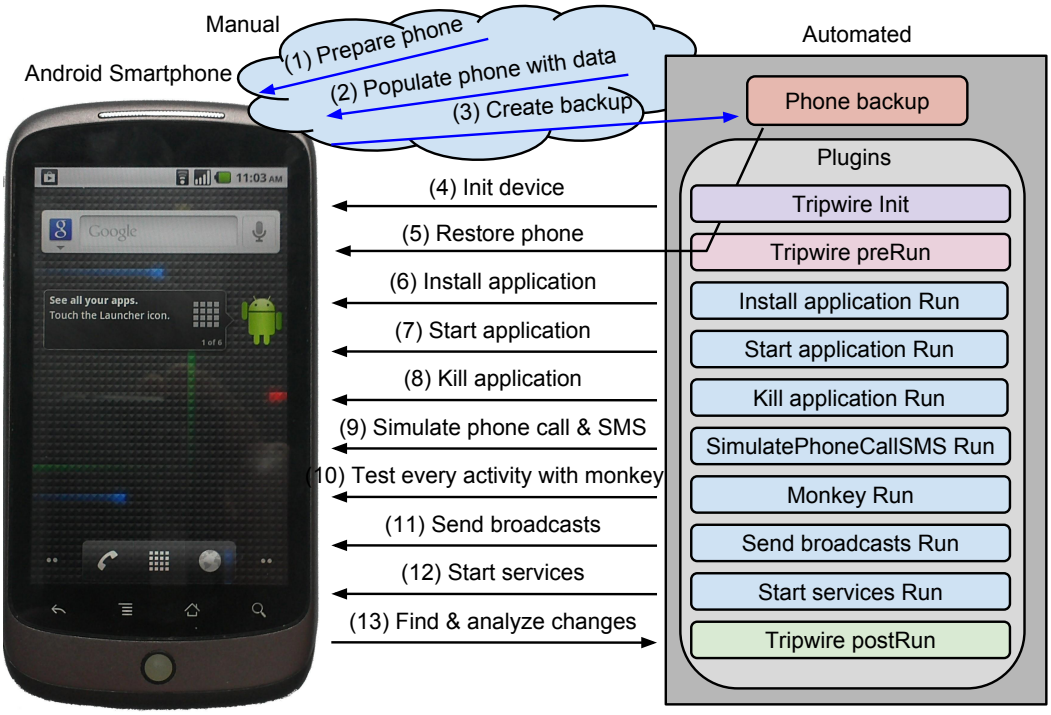

Figure 5.2: The analysis sequence of the *Tripwire Analysis*.

Chapter 6

Results

This chapter presents the results of the *DroidBox* and *Tripwire Analysis*. As both *Analyses* aim to detect malware, they were evaluated against the malware samples from the *Android Malware Genome*[1] project [16]. The *Android Malware Genome* project contains samples from 49 different malware families including root- as well as non-root malware. The remaining chapter is structured as follows. Section 6.1 presents the combined detection results of both *Analyses*. Section 6.2 and Section 6.3 present the results of the *DroidBox Analysis* and the *Tripwire Analysis* respectively, in more detail.

6.1 Combined Results

Figure 6.1 summarises the combined evaluation results of both *Analyses*. The x-axis represents the percentage of samples that were detected, grouped by malware family. The green colour shades mark detected samples, and the red colour stands for unrecognised samples. The different shades of green define which *Analysis* marked the respective sample as suspicious. The y-axis lists the various malware families. Most noticeable is that some malware families are not detected by any of the *Analyses*. Nearly all remaining malware families are almost exclusively detected by one of the two *Analyses* and not the other one. The only exception is the *GingerMaster* malware family. This framework marked 510 of the 1260 analysed samples as suspi-

[1] http://www.malgenomeproject.org

cious, which leads to a detection rate of 39.8%. Google Bouncer was tested against the same samples and achieved at the end of 2012 a detection rate of 15.32% [80]. Figure 6.1 clearly shows that some families[2] are detected very well, others[3] have a detection rate near zero or zero and the remaining families contain applications that are detected, as well as applications that did not behave suspicious during the evaluation. Through the high amount of applications, it was not possible to investigate all of them in detail. Instead, some randomly picked applications were analysed manually to learn why they were detected or why they were not detected. The results of the manual analysis are grouped by malware family.

ADRD: This malware family steals personal data, like the IMEI and IMSI and sends it to a remote server. A closer look was taken on the repacked sample with the package name "com.olivephone.cu"[4]. The application tries to send personal data to two remote locations[5]. This sample was not marked as suspicious because both addresses were unreachable. Therefore, the application was not able to transmit the data. For more details on this malware family refer to [81].

BaseBridge: This malware tries to gain root privileges and to consequently install a malicious application. The application to be installed is included as resource (res/raw/anserverb) in the APK of the *BaseBridge* sample [82, 83]. If the installation succeeds the second application tries to contact a Command & Control (C&C) server. The malware can send short messages, dial phone numbers, and remove incoming short messages. The analysed sample[6] contains the root exploit as well as the app to be installed in the */assets* folder. The service that executes the exploit and consequently installs the application was called during the automated analysis. It is unclear why the exploit or the

[2]AnserverBot, Bgserv, Dogwars, DroidCoupon, DroidDream, FakePlayer, Gingermaster, GoldDream, Gone60, HippoSMS, JiFake, jSMSHider, Plankton, SMSReplicator, Walkinwat, and Zsone

[3]ADRD, BeanBot, CoinPirate, CruseWin, DroidDeluxe, DroidDreamLight, DroidKungFu3, DroidKungFu4, DroidKungFuUpdate, Endofday, FakeNetflix, GamblerSMS, Geinimi, GGTracker, GPSSMSSpy, KMin, LoveTRap, NickyBot, NickySpy, RogueLemon, PogueSPPush, SndApps, Spitmo, Tapsnake, YZHC and Zitmo

[4]SHA1 fingerprint: 7ded7bb7041acce78e85e811c0ac048338bdc2d9

[5]`http://adrd.xiaxiab.com/pic.aspx` and `http://adrd.taxuan.net/index.aspx`

[6]SHA1 fingerprint: 4de1730332ac35e99337c78ab9aee4fc93f71fc0

Figure 6.1: This figure shows the detection results of the *DroidBox* and *Tripwire Analysis* grouped by malware families.

installation failed. The exploit, although it is on the blacklist, was not detected by Tripwire because the resources are not extracted. Instead, they are loaded directly from the APK [84]. However, this malware can be easily detected using static analysis techniques.

BeanBot: This malware connects to a C&C server and retrieves instructions [85]. The server can e.g. instruct the client to send short messages to premium numbers. During the analysis, the address of this server was not found. Therefore, it was not possible to determine if it is still running. No suspicious behaviour was monitored during the dynamic *Analysis*.

CruseWin: This malware[7] turns the smartphone into a bot. The application gets its commands from a C&C server[8] in XML form. The C&C server can, for example, instruct the smartphone to send or forward short messages or to uninstall applications [86]. During the analysis, the C&C server was not working and therefore the application did nothing suspicious.

DroidKungFu 3: Two samples were analysed, the first sample[9] was marked as suspicious during the automated analysis, the second sample[10] not. Both samples include two encrypted root exploits and an encrypted application [87]. The encrypted application is decrypted and subsequently installed after the malware gains root privileges. The installed application connects to a C&C server and waits for commands. The first sample copied a binary into the "/system" folder and installed the application. It is unclear why the second sample did nothing suspicious.

DroidKungFu 4: The analysed sample[11] was marked as suspicious on the rooted device and the device with the modified *su* binary. However, it was not marked as suspicious on the unrooted device. The reason is, this sample (family) does not include a root exploit. Instead, it asks for root permissions, which failed on the unrooted device. If this sample obtains root privileges, it copies several

[7] SHA1 fingerprint: 438e0b566eca22e7168711931a958736d9a50118
[8] `http://crusewind.net/flash/test.xml`
[9] SHA1 fingerprint: 255a1b74428b5615d65f39775ec7234e27bd9e74
[10] SHA1 fingerprint: 0a51a11062bf106b10500cab45de1b091af2a06f
[11] SHA1 fingerprint: 971b3df7ea0f9134b69a1c1f89bbd09d085ca855

files into the "/system" folder. These binary files try to connect to an C&C server to receive instructions.

Geinimi: This malware family has botnet-like capabilities. The C&C server can instruct the client to do a variety of things, like installing an application, sending a short message, or uploading information to the server [88, 89]. The C&C server URLs are stored encrypted in the application. Researchers [88] from Lookout decrypted a sample and found several C&C server URLs. This sample decrypted a different URL[12] and tried to connect to it, during the automated analysis. This URL and the other URLs mentioned in the related work were not working at the time of the manual analysis. According to the *Analysis* log, it seems that the server did not respond during the dynamic analysis.

KMin: This malware family collects information on the device and sends it to a remote server [90]. The collected information consists of the IMEI, the IMSI, and the current time. The analysed sample[13] collected this information and sent it to the remote server[14]. It was not marked as suspicious, because the accumulated *suspicious score* for these two events is only six. Nearly all automatically analysed samples showed the same behaviour. Increasing the score for these events would lead to many false positives, because many advertisement libraries use this information too.

SndApps: Applications belonging to this family can be categorized as Spyware. The application registers a receiver for the *BOOT_COMPLETED*[15] broadcast (c.f. 2.1.1) which starts a service. This service steals private data, such as the e-mail address, the carrier or the *deviceId* and sends it to a remote location[16]. There was no network traffic monitored to this server during the *Analysis*.

YZHC: This malware connects to a remote server to fetch premium rated numbers. Consequently, it sends short messages to these numbers. Incoming messages

[12] http://www.winpowersoft.com:8080/adserver/getAdXml.do
[13] SHA1 fingerprint: 005ce595935bf1aae4d53ab4bbd92ce7c81e5b2a
[14] http://su.5k3g.com/portal/m/c5/0.ashx
[15] android.intent.action.BOOT_COMPLETED
[16] `http://www.typ3studios.com/android_notifier/notifier.php`

that inform the user about these services are deleted [91]. The analysed sample[17] was marked as suspicious because it called the number "10086". This number belongs to the customer service portal of China Mobile. Probably this event was triggered by *Monkey*. The C&C server[18] was not working at the time of the analysis. This is the reason why the malware did not send short messages to a premium rated numbers.

Zitmo: Only one sample[19] of this malware family was available. Therefore this sample was analysed. This malware forwards incoming short messages together with the *deviceId* to a remote server[20]. Normally it should be easy to detect this kind of malware with DroidBox. It was not detected because the remote server did not exist anymore, and therefore this malware was not able to leak sensitive information.

Zsone: A closer look was taken on the sample[21] that was not detected. It turned out that the sample contains malicious code, such as sending short messages to "10621900", "10626213", and "106691819", but this code is never called. A possible explanation could be, that the malicious code was automatically added to the benign application and that the patching process failed to add a call to the malicious code.

The results show that both *Analyses* have their respective advantages and disadvantages. The *DroidBox Analysis* is well suited to detect malware that leaks sensitive information or produce costs for the user. The *Tripwire Analysis* has its strength at detecting malware which use permanent root exploits. Both *Analyses* reveal the weak points of dynamic analyses. Only one execution path is analysed. If the malicious code is not triggered for some reason, or the execution of the malicious code fails, the dynamic analysis can not examine it. During the manual analysis, two main reasons were found for the failure of the execution of the malicious code. Many samples tried to connect to an C&C server that was not working. These samples

[17] SHA1 fingerprint: 7e04d1854382dbb42417ab4e5eab142ebb482ff9
[18] http://domaindev.51widgets.com/ss/dom/config.xml
[19] SHA1 fingerprint: c9368c3edbcfa0bf443e060f093c300796b14673
[20] `http://softthrifty.com/security.jsp`
[21] SHA1 fingerprint: 14f5c14af60b5930f9dfbeed30f5529ba814c0e6

were not able to leak sensitive data or to receive commands. Other samples failed to use their exploits. This can happen through unforeseen circumstances or through security updates that fix the exploitable code. A general problem is how to decide whether an event should be rated as suspicious or unsuspicious. It is very difficult to decide this automatically, because it depends on the context. Thus, the framework does not label samples as benign or malicious, but gives them a *suspicious score*. The following sections present the analysis results of the respective *Analyses* in more detail.

6.2 DroidBox

This section presents the results of the *DroidBox Analysis*. The *Analysis* was executed in a VM running a 32-bit Ubuntu 12.04. The host machine was an off-the-shelf Personal Computer (PC) with an Intel Core 2 Quad CPU Q9550 and 8GB RAM. To speed up the *Analysis*, three instances of the VM did run in parallel. The samples were analysed with DroidBox 2.1 and DroidBox 2.3, each with and without access to the Internet. Figure 6.2 shows the combined detection rate of all four DroidBox runs, broken down to the 49 analysed malware families. Samples belonging to the families *AnserverBot*, *Bgserv*, *DogWars*, *FakePlayer*, *GingerMaster*, *GoldDream*, *Gone60*, *HippoSMS*, *JiFake*, *Plankton*, *SMSReplicator*, *Walkinwat*, and *Zsone* were detected quite well, but the overall detection rate over all families with DroidBox is only 29.4%. Figure 6.3 shows the runtime of the *Plugins* in detail. Most noticeable is the very long maximal runtime of the *Monkey Plugin*. It is caused by applications with many activities. It was a goal to start and test every activity with *Monkey*. To limit the execution time, *Monkey* tests the first launcher activity for at most 10 minutes and every other activity from one to three minutes or 300 events depending on the application's activity count. The next noticeable thing is that the maximal runtime for most *Plugins* is about 5 minutes. That is because from time to time *adb* commands hang, and the default kill time was set to five minutes. Figure 6.4 shows the sum of all suspicious events that happened during the execution of the respective *Plugin*. The *Plugin* order in the figure equals the *Plugin* order during the *Analysis*. The *Monkey Plugin* triggered the most suspicious actions, followed by

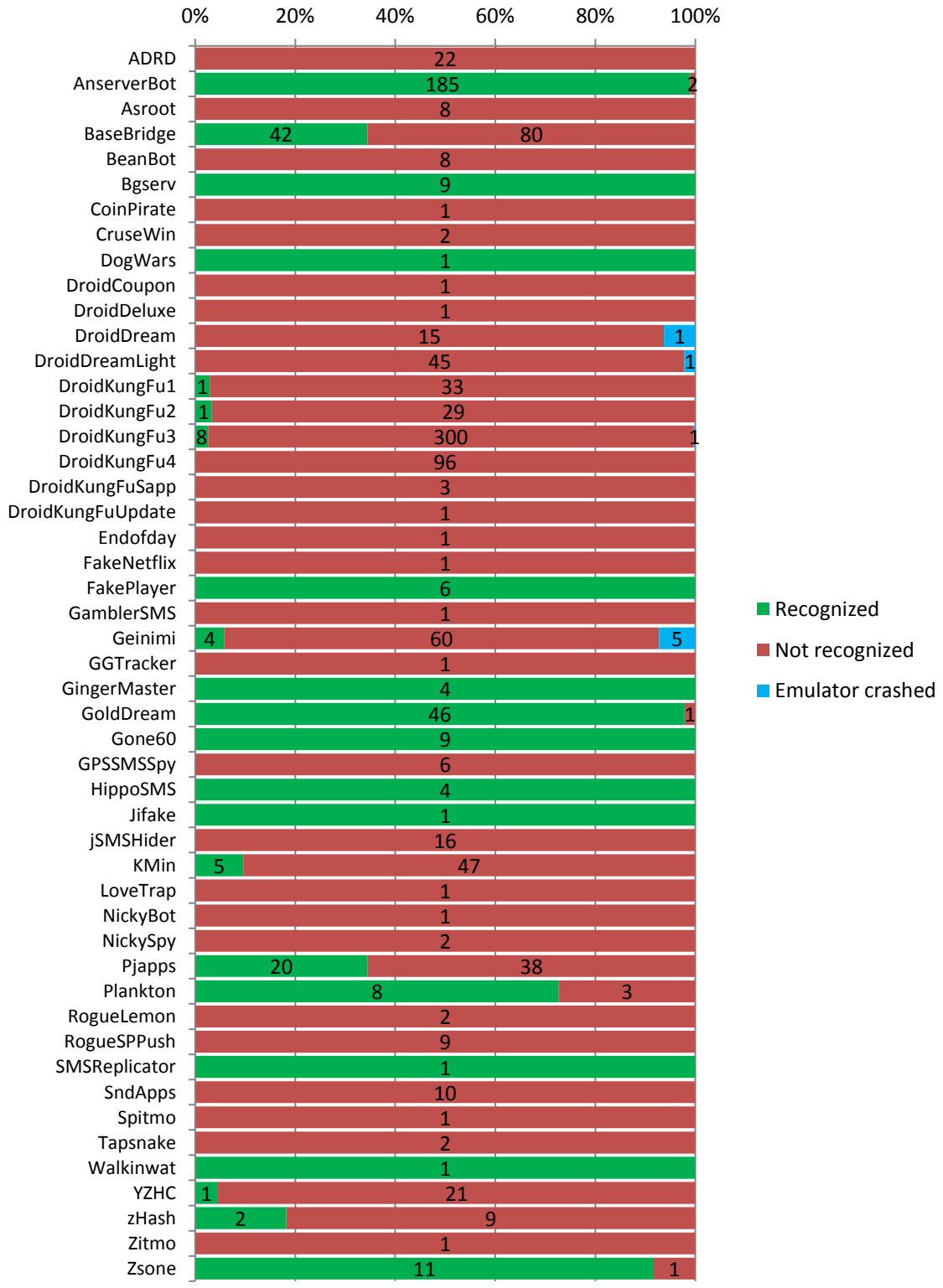

Figure 6.2: This figure shows the merged detection results of all four DroidBox runs.

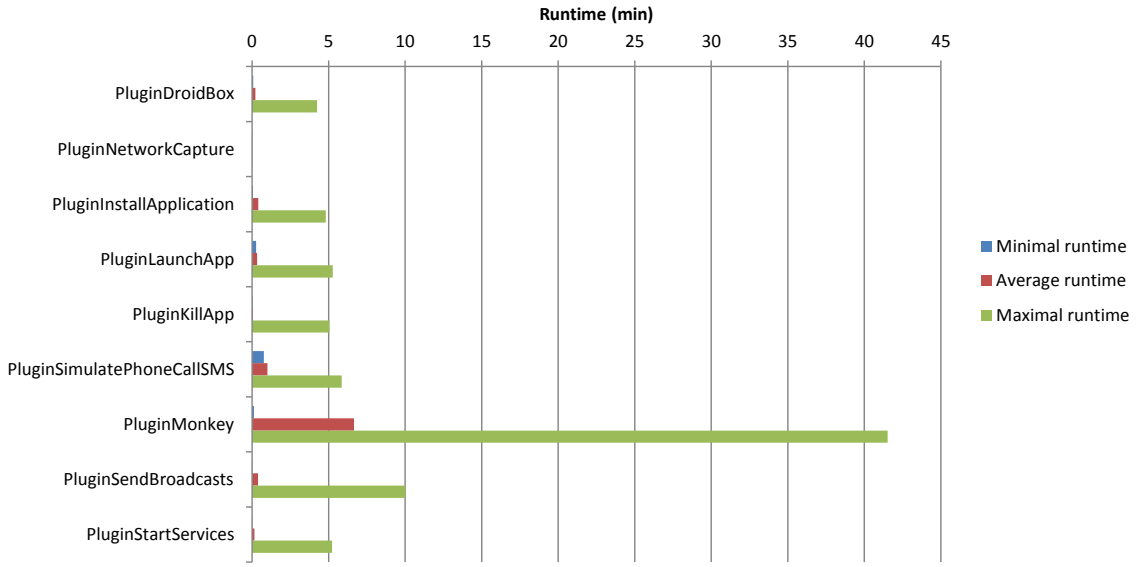

Figure 6.3: This figure shows the runtime of all *Plugins* executed during the *DroidBox Analysis*.

the *LaunchApp Plugin*, the *SendBroadcasts Plugin* and the *SimulatePhoneCallSMS Plugin*. Figures 6.5 and 6.6 show the *suspicious score* over time while the *Mon-*

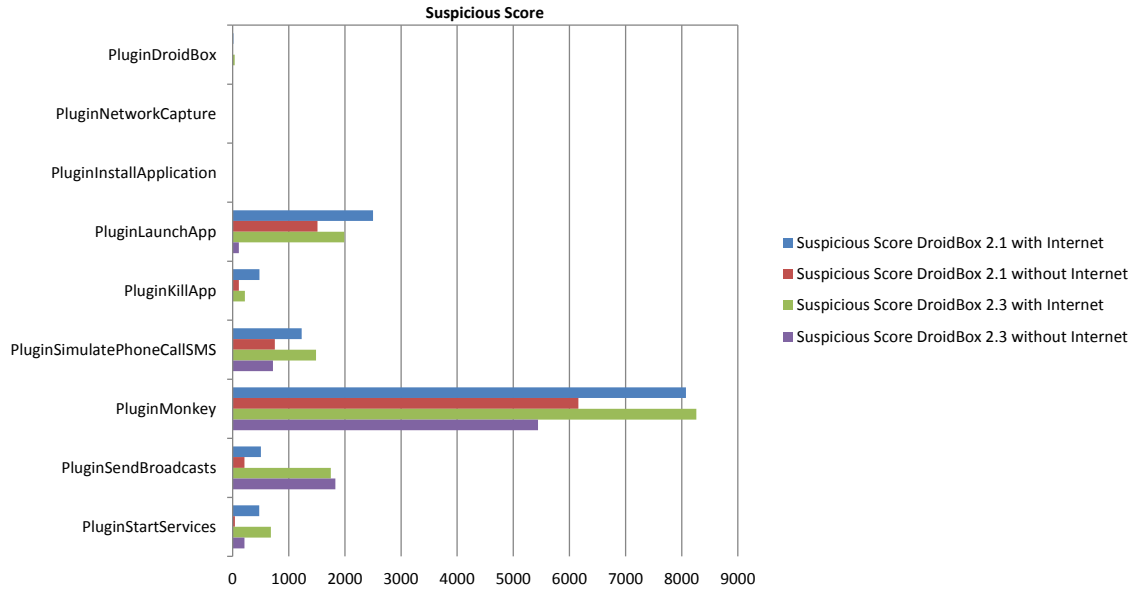

Figure 6.4: This figure shows the sum of all *suspicious scores* that occurred during the execution of the corresponding *Plugin*.

key Plugin is running. Figure 6.5 shows the *suspicious score*, over all activities, from start to end of the *Monkey Plugin*. Figure 6.6 shows only the *suspicious score* while Monkey was executed on the first launcher activity. These figures illustrate that most of the suspicious events happen directly after the start-up of the applications. The next notable property is that almost no new suspicious event occurred after about 10 minutes. This means that the overall runtime of the *Monkey Plugin* could be limited to 10 minutes, for the analysed malware, without losing important information.

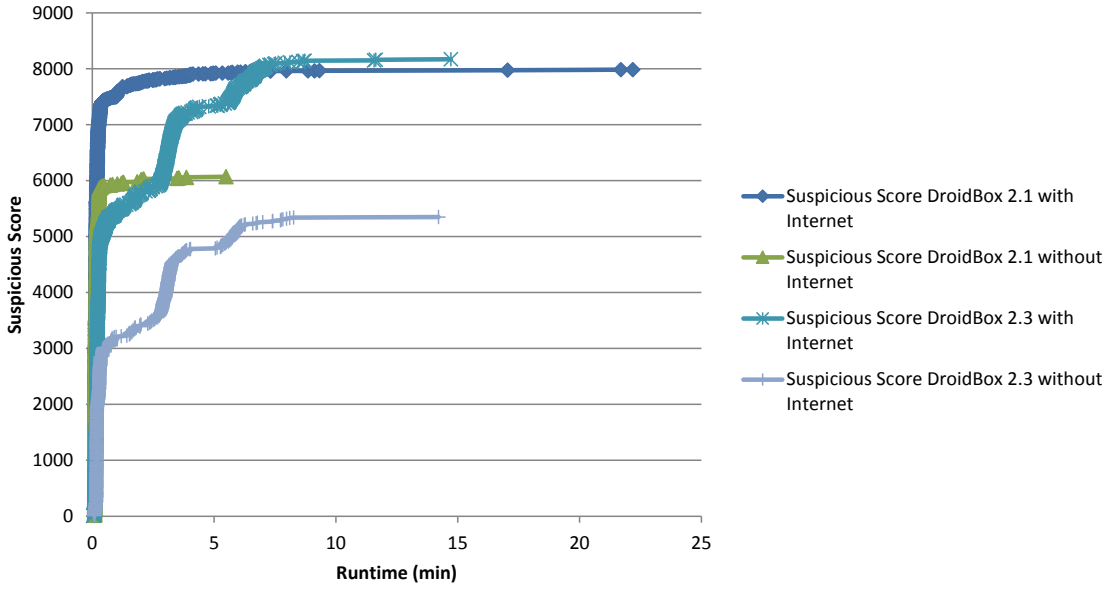

Figure 6.5: This figure shows the suspicious events over time during the execution of the *Monkey Plugin* on the Malgenome samples.

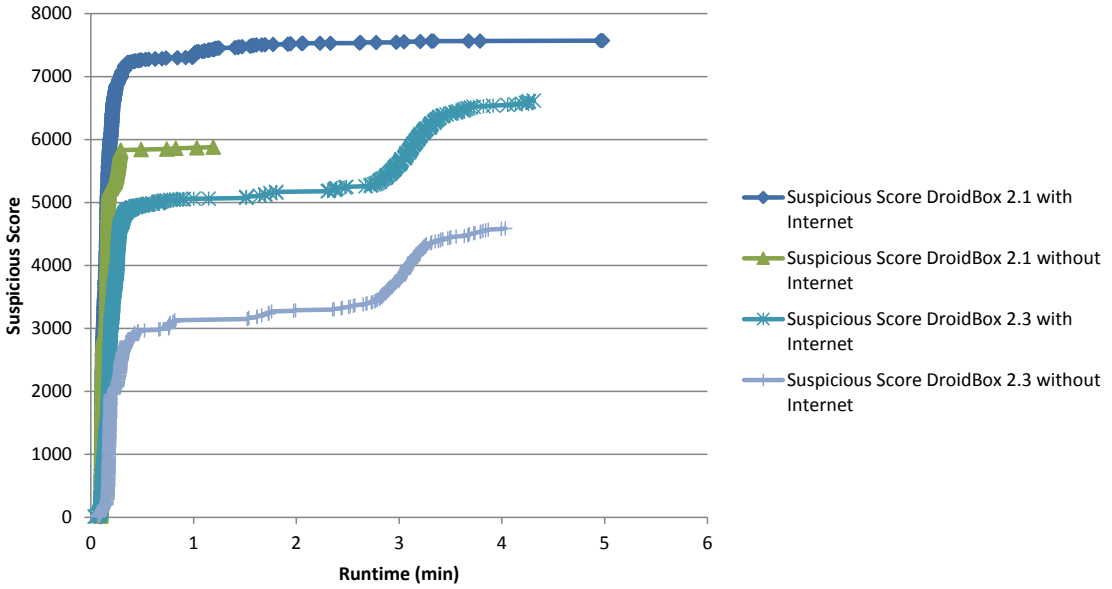

Figure 6.6: This figure shows the suspicious events over time during the execution of *Monkey*. In contrast to the previous figure, this figure covers only the first run of *Monkey* on the main screen of the analysed applications.

6.2.1 Top Free Applications

As the Malgenome project contains only malicious applications, some benign applications were needed to evaluate the false positive rate. Therefore, the top applications of all categories from the official *Google Play Store* were downloaded. The download happened on November 19, 2012 and incorporated 1282 applications. These applications were analysed with and without Internet access. Figure 6.7 shows the results, from the run with Internet access. The most eye-catching thing is that a high amount of applications failed to install.Most installations failed because of missing shared libraries. This is most probably caused by the missing Google Maps API on the emulator. The next notable thing is the very low number of applications that are marked as suspicious (green). The suspicious applications were analysed in more detail to learn why they were marked as suspicious.

- at.fhooe.mc.app

  This application is called “Rotes Kreuz” and aims to help in emergency situations. It explains first aid measures and allows to directly call several emergency telephone numbers. It was marked as suspicious because it called the

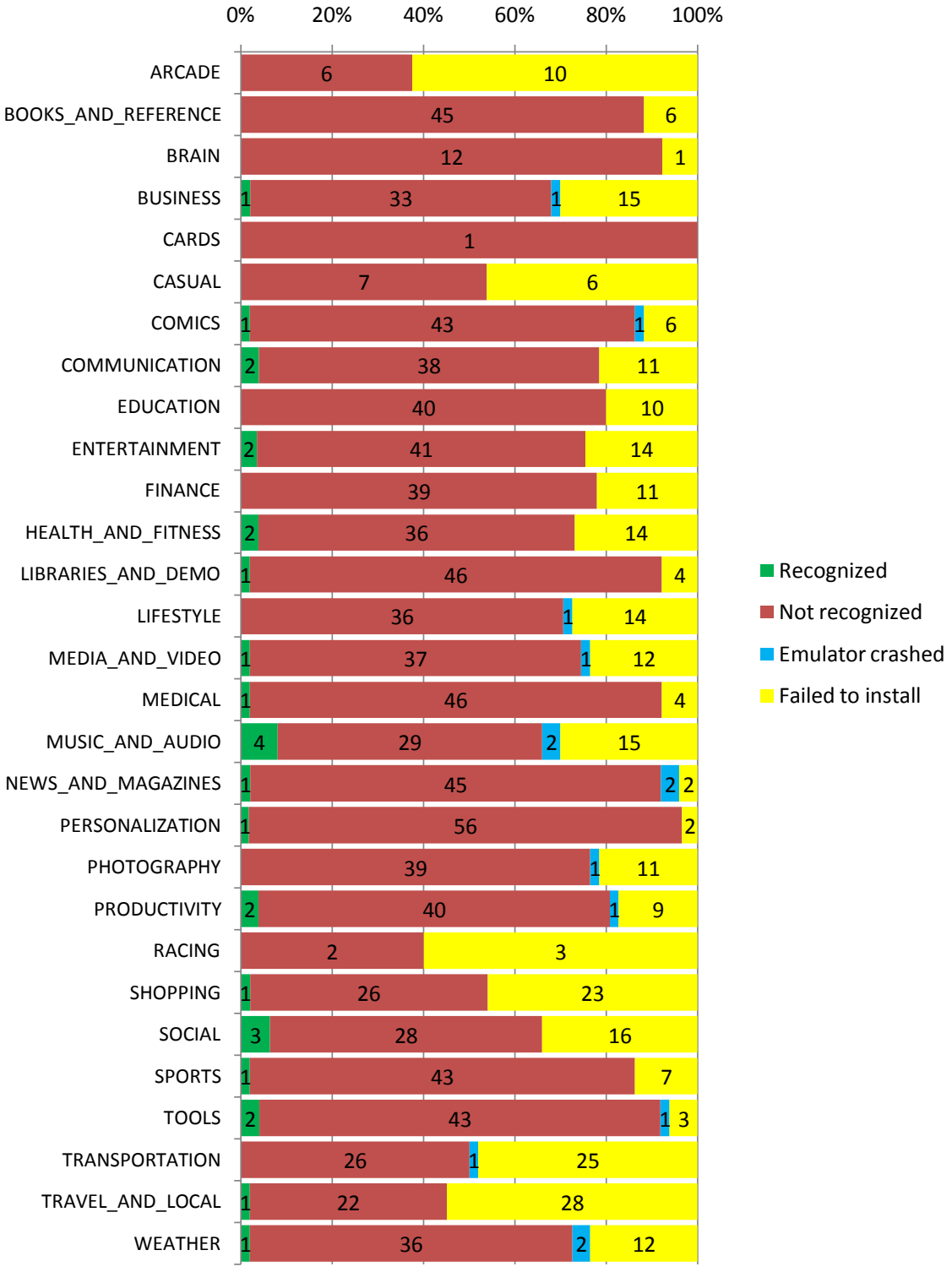

Figure 6.7: This figure shows the detection rate (false positives) of DroidBox along the Top Free Google Play Store applications.

following (emergency) phone numbers:

  - 112 (European emergency number),
  - 122 (fire brigade),
  - 128 (gas emergency),
  - 140 (mountain rescue),
  - 141 (doctor's emergency line),
  - 144 (emergency medical services),
  - 1455 (pharmacy emergency number),
  - 116123 (bank card emergency service),
  - +43/1589000 (blood donor centre) and
  - +43/14064343 (poison information centre)

  It would be easy to whitelist these numbers, but then calls initiated by malicious code would not be analysed.

- at.mobilkom.android.handyparken
  This application was marked as suspicious because it sent two messages "A30" and "Nein" to +4382820200 (parking ticket number). The application allows to buy car park tickets via short messages.

- com.avast.android.mobilesecurity, com.symantec.mobilesecurity and com.tf.thinkdroid.amlite
  The applications "avast! Mobile Security", "Norton Security & Antivirus" and "ThinkFree Office Mobile Viewer" were marked as suspicious because they use the *DexClassLoader* function.

- com.jb.gosms
  This application is an alternative messaging app. It was marked as suspicious because three short messages were sent during the analysis. These messages (":O", "" and "6o") look like they were generated by pseudo-random events caused by the *Monkey Plugin*.

- de.huwig.rhok.notfall.apk
  This application aims also to help in emergency situations. It was marked as suspicious because it called the following phone numbers: +49/8001110111 (crisis line) and +49/55119240 (poison information centre).

- nao.parkscheinpro
  The application sent several messages to the number 06646606000 (parking ticket number). The application allows to buy car park tickets via short messages.

- The remaining suspicious applications[22] were marked as suspicious because DroidBox reported that tainted data (TAINT_SMS) were sent to a remote server. Manual checks of some randomly picked applications suggest that all these application were marked incorrectly.

For most applications it is understandable why they were marked as suspicious, only the "'TAINT_SMS"' information seems to be unreliable.

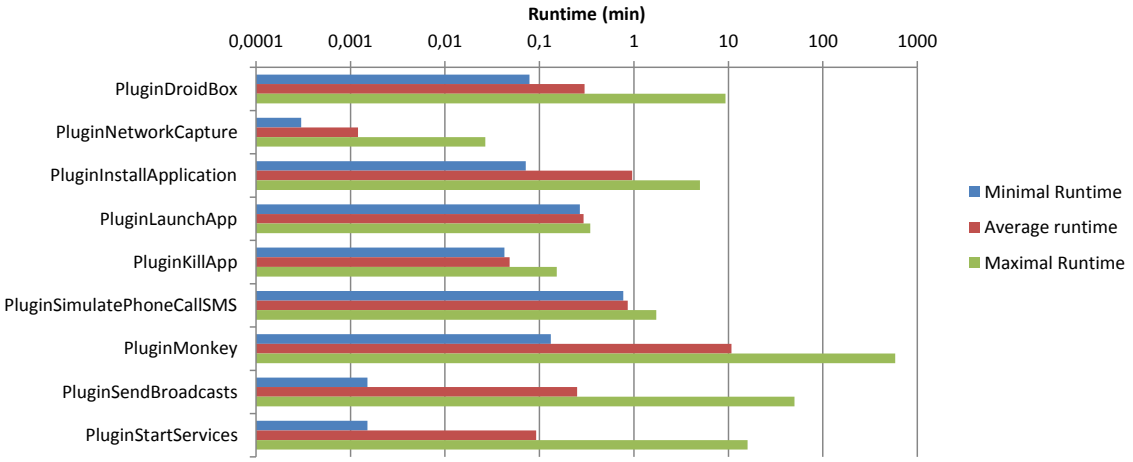

Figure 6.8: This figure shows the runtime of the different *Plugins* executed during the *DroidBox Analysis* of the Top Free applications.

[22] com.aol.mobile.engadget, com.aportela.diets.view, com.bianor.ams, com.bubblesoft.android.bubbleupnp, com.estrongs.android.safer, com.headcode.ourgroceries, com.linkedin.android, com.mobilityflow.animatedweather.free, com.n7mobile.nplayer, com.netqin.mobileguard, com.pompeiicity.funpic, com.scoompa.facechanger, com.skout.android, com.tumblr, de.komoot.android, hr.podlanica, kik.android, laola.redbull, mobi.lockscreen.magiclocker, tunein.player, tv.dailyme.android

Figure 6.8 shows the *Plugin* runtime. Due to the huge difference in the magnitudes, which range from less than one minute to about 600 minutes, a logarithmic scale was chosen for the time axis. As expected, most *Plugins* finish quick, the exception is again the *Monkey Plugin*. Figure 6.9 shows the suspicious events over time during the execution of the *Monkey Plugin*. It confirms that for the analysed applications the very long runtime of the *Monkey Plugin* could be limited to about ten minutes without losing relevant information. As expected more suspicious events are detected with Internet access. Figure 6.10 shows the suspicious events per *Plugin*. Again the

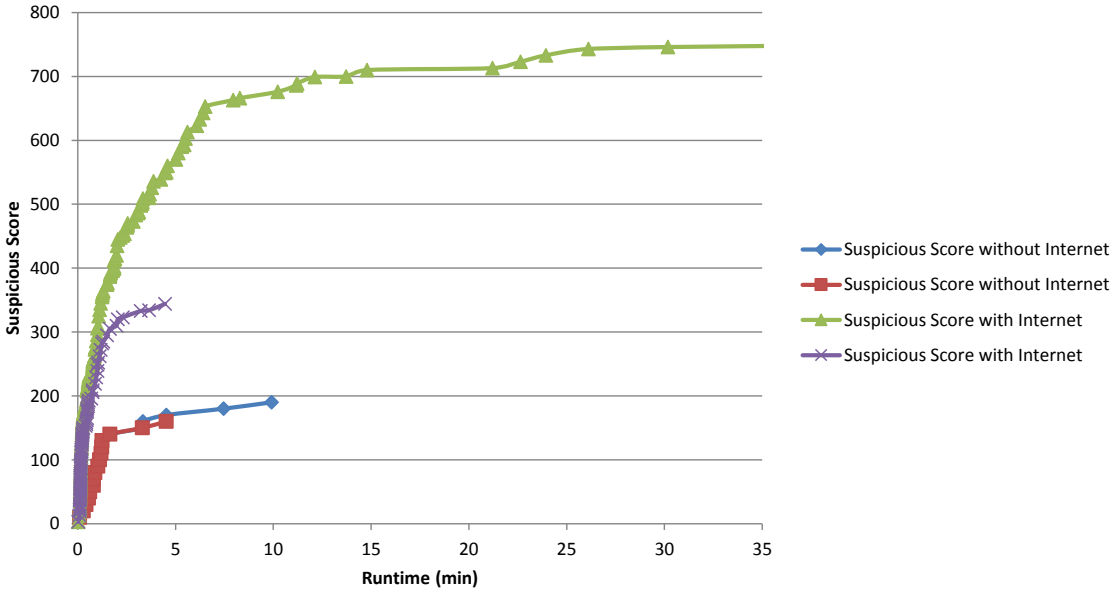

Figure 6.9: This figure shows the suspicious events over time during the execution of the *Monkey Plugin* on the Top Free applications.

Monkey Plugin triggered the most suspicious events. Compared to Figure 6.4 it is noticeable that the *LaunchApp Plugin* and the *SendBroadcasts Plugin* triggered much less suspicious events when executed on benign applications. This means that benign applications do much less suspicious things without a user (or *Monkey*) interaction.

The *DroidBox Analysis* is well suited to detect malware that leak sensitive information or produce costs for the user. The best results were achieved with Internet access. This can easily be explained, without Internet access malware can

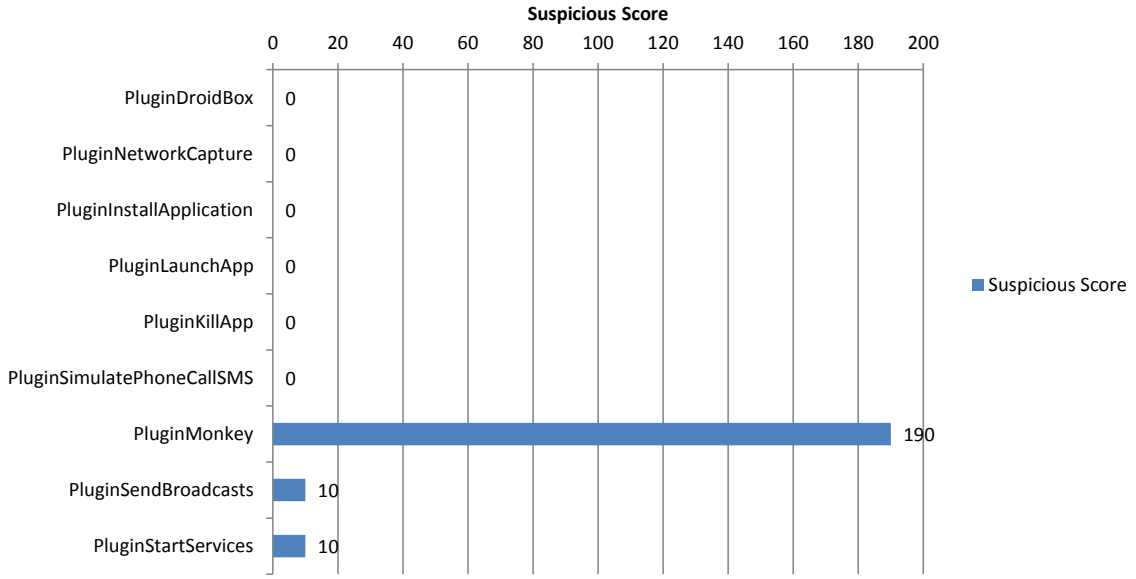

Figure 6.10: This figure shows the suspicious events that occurred during the execution of the *DroidBox Analysis*, on the Top Free applications.

not contact their C&C servers. Furthermore, Internet access provides an easy way to leak sensitive data. The DroidBox system provides accurate results, only the "TAINT_SMS" tag seems to be unreliable.

6.3 Tripwire

The *Tripwire Analysis* aims to detect applications that use root privileges, by analysing file-system changes after the execution of the suspicious application. The *Analysis* was executed on a Nexus One running Android 2.2. The smartphone was attached to a Chromebook with Ubuntu, running the analysis framework. To see if it makes a difference whether the smartphone is rooted or not, all samples were executed three times one the Nexus One with different system configurations. One system was non-rooted, one was rooted and one was rooted and had a modified "su" binary. The non-rooted phone was created by deleting the "su" binary and the "Superuser.apk". The phone with the modified "su" binary grants root access to everyone and creates a file that documents the use. At all runs, the device had a SIM card inserted and was connected to the Internet. The next sections present

the initial and the final results.

6.3.1 Initial Results

Figure 6.11, 6.12 and 6.13 show the first outcomes of the *Tripwire Analysis* on the rooted, modified rooted and unrooted phone respectively. These figures show only the Malware families that are known to use privilege escalation techniques. The results of all three runs are very similar, except for the *GingerMaster* and the *DroidKungFu*{*1, 2, 3, Sapp*} Malware-families. The *GingerMaster* family shows suspicious behaviour only on the non-rooted phone. In contrast, the *DroidKungFu*{*1, 2, 3*} families show less suspicious behaviour on the non-rooted phone. The *DroidKungFuSapp* family did not show suspicious behaviour when executed on a phone with a modified "su" binary. The detection rates are 12.77%, 11.72% and 8.06% on the rooted, modified rooted and unrooted phone respectively.

Figure 6.11: The initial detection rate of the *Tripwire Analysis* on a rooted phone with Internet access against the samples from the Malgenome project was 12.77%.

6.3.2 Improved Results

In order to improve the detection rate, the file-system changes from randomly selected samples were analysed in detail. Furthermore, individual samples were executed again but this time stimulated by hand. This was done by using the *User Plugin* instead of the *Monkey Plugin*. For example, one of the manual analysed

Figure 6.12: The initial detection rate of the *Tripwire Analysis* on a phone with a modified su-binary and Internet access against the samples from the Malgenome project was 11.72%.

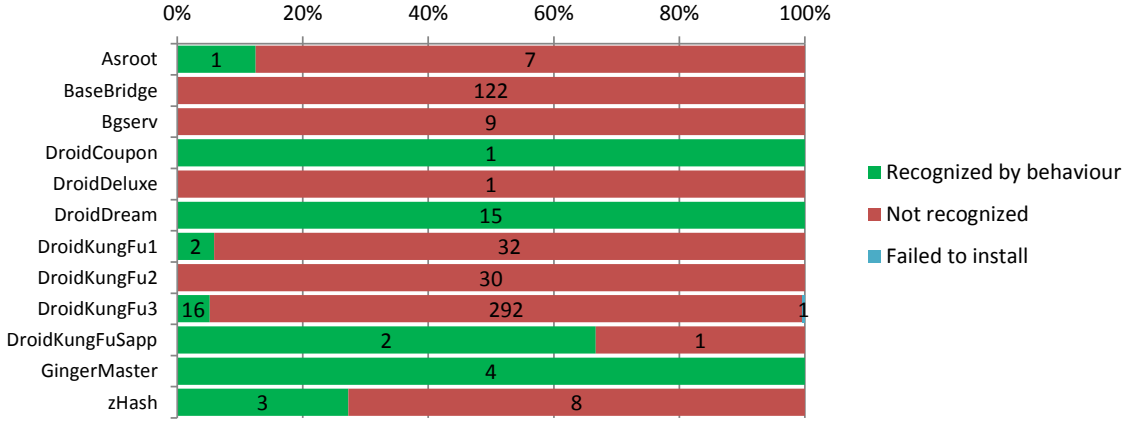

Figure 6.13: The initial detection rate of the *Tripwire Analysis* on an unrooted phone with Internet access against the samples from the Malgenome project was 8.06%.

applications[23] a sample belonging to the *DroidDeluxe* family showed an interesting behaviour. This sample changed the permission of specific files to world-readable and world-writable. The affected files are shown in Listing 6.1. These files contain sensitive information such as contacts, account information and messages. By default, these files have the permission `-rw-rw----`, meaning that only the application that owns the data can access them. After the execution of the application,

[23]SHA1: 8e2f43e46335b8a4ce68c920660def6e9c14c712.

```
/data/data/com.android.providers.contacts/databases/contacts2.db
/data/data/com.android.providers.settings/databases/settings.db
/data/data/com.android.providers.telephony/databases/mmssms.db
/data/system/accounts.db.
```

Listing 6.1: Files with changed permissions (world readable & writable) after the execution of a *DroidDeluxe* sample.

```
path=/data/system/shared_prefs/log_files.xml permission changed from rw------- to↩
    -rw-rw----

path=/data/system/packages.xml permissions changed from -rw------- to -rw-rw-r--
```

Listing 6.2: False positive detected files

the permissions were changed to `-rwxrwxrwx`, which means everyone can access the files. The contents were not modified. During the three automated runs this behaviour could not be observed, and even during the manual tests this behaviour occurred very rare. Adding a rule that searches for files with changed permissions, changed owner or changed group did not improve the detection results. The rule finds "suspicious files" (see Listing 6.2), but these changes occur independently of the analysed application and therefore lead to false positive results. Therefore, this rule was deactivated, changes in critical system folders are anyway detected by a different rule.

During the manual analysis of the changes, it was conspicuous that several applications contain root exploits but were not marked as suspicious. The reason for this is, that the owner and the group of the files were the respective applications and the files were stored in the applications folder. Without looking at the content it is not possible to decide whether these files are suspicious or not. To detect these files as well, the *Tripwire Plugin* was extended to create and maintain a list of suspicious files, more precise of their fingerprints. The fingerprint of a file allows to find copies of the file effectively. Every time a suspicious file is detected, the *Tripwire Plugin* adds the fingerprint of the file to the list of suspicious files. If the same file is detected in a future *Analysis*, it will be marked as suspicious even if it has no other suspicious property.

Another attempt to improve the detection rate was to search for duplicated files.

```
path=/data/data/com.gmail.nagamatu.drocap/files/camera_click.ogg, permissions=-r↩
    w-rw-rw-, owner=app_54, group=app_54
copy of:path=/system/media/audio/ui/camera_click.ogg, permissions=-rw-r--r--, own↩
    er=root, group=root, hash=8deb78656237403b094fc4f762836bf601f53547

path=/data/data/cmp.LocalService/files/DATA_Preferences, permissions=-rw-rw----, ↩
    owner=app_54, group=app_54
copy of:path=/data/data/com.google.android.location/files/DATA_Preferences, permi↩
    ssions=-rw-rw----, owner=system, group=system, hash=c66bf56ddabc2021b84d3a↩
    e2755d0ab05ff0c99e

path=/data/data/org.jiaxxhaha.netraffic/shared_prefs/_has_set_default_values.xml,↩
     permissions=-rw-rw----, owner=app_54, group=app_54
copy of:path=/data/data/com.android.phone/shared_prefs/_has_set_default_values.xm↩
    l, permissions=-rw-rw----, owner=radio, group=radio, hash=9d466c28c76a77c06↩
    f262152705d01e952eac811

path=/data/misc/bluetooth/dynamic_auto_pairing.conf, permissions=-rw-------, owne↩
    r=system, group=system
copy of:suspect=0, path=/system/etc/bluetooth/auto_pairing.conf, permissions=-rw-↩
    r-----, owner=system, group=system, hash=15298cfee937dc6eb71ba3536bd9c147b6d5↩
    aebf

path=/data/data/com.alan.siwameinv7/databases/webviewCache.db, permissions=-rw-r↩
    w----, owner=app_54, group=app_54, hash=463af814f54b5afe990748d1f37c0627410dc↩
    e12
copy of:suspect=0, path=/data/data/com.google.android.gm/databases/webviewCache.d↩
    b, permissions=-rw-rw----, owner=app_50, group=app_50, hash=463af814f54b5af↩
    e990748d1f37c0627410dce12

path=/data/data/org.drhu.waterdropletfree/databases/webview.db, permissions=-rw-r↩
    w----, owner=app_54, group=app_54, hash=3a5fc8b4f24ba8652096a24954f06229dcf3e↩
    ded
copy of:suspect=0, path=/data/data/com.android.vending/databases/webview.db, perm↩
    issions=-rw-rw----, owner=app_15, group=app_15, hash=3a5fc8b4f24ba8652096↩
    a24954f06229dcf3eded
```

Listing 6.3: Examples of false positive detected files

The idea was to detect applications that copy system files or other sensitive files from the system or other applications. Therefore, all added or changed files are compared against the files, from the initial snapshot. This approach led to many false positives. Some of the files that have led to false positives are shown in Listing 6.3. These findings are not useful for root malware detection, because these files are even copied by benign applications without root access. It is even hard to tell which files were copied, because some of the files just contain an XML definition and an empty map or a basic database scheme. Nevertheless it could be interesting to take a closer look at some of the findings. For example, many applications copied databases from different applications. The last two findings in Listing 6.3 are examples of copied databases. The “webviewCache.db” was copied from the Gmail

```
Copied file: suspect=3, path=/data/data/cn.buding.coupon/files/busybox, permissio↩
    ns=-rwxrwxrwx, owner=root, group=app_54
copy of:suspect=0, path=/system/xbin/echo, permissions=lrwxrwxrwx, owner=root, gr↩
    oup=root, hash=f57c5db795dfb323c157bad2517199ffe3c9f135

Copied file: suspect=3, path=/data/data/cn.buding.coupon/files/busybox, permissio↩
    ns=-rwxrwxrwx, owner=root, group=app_18
copy of:suspect=0, path=/system/xbin/mkswap, permissions=lrwxrwxrwx, owner=root, ↩
    group=root, hash=f57c5db795dfb323c157bad2517199ffe3c9f135

Copied file: suspect=3, path=/data/data/com.igamepower.appmaster/files/sh, permis↩
    sions=-rws--x--x, owner=root, group=root
copy of:suspect=0, path=/system/bin/sh, permissions=-rwxr-xr-x, owner=root, grou↩
    p=shell, hash=dd954c535f9a5851f028151fbaab30495545c8a5
```

Listing 6.4: Examples of files detected by searching for duplicated files.

application, the "webview.db" database was copied from the Google Play Store application. These and other databases could potentially contain sensitive data [92]. For lack of time, these findings were not analysed.

After filtering the approach yields interesting results. Some examples are shown in Listing 6.4. Most of the findings of the improved rules were already detected by a different rule. Therefore, the detection rate improved only slightly. Figures 6.14, 6.15 and 6.16 show the improved detection results. The improved detection rates are 15.9%, 14.8% and 11.4% on the rooted, modified rooted and unrooted phone respectively. Compared to the initial results the detection rate for the Asroot, DroidDream, DroidKungFu1, DroidKungFu2 and DroidKungFu3 family improved. As before the rooted phone yields to the best detection results. In contrast to the initial results, the legend differentiates now between three different recognition types. Green marks apps that are recognised by behaviour. Dark green marks apps that show no suspicious behaviour but contain at least one file that is blacklisted. Light green marks apps that show suspicious behaviour and contain at least one blacklisted file. Figure 6.17 shows the runtime of the distinct *Plugins*. Because of the huge runtime differences the time-axis is scaled logarithm. The average runtime of all *Plugins* is below ten minutes. The *Tripwire Plugin* and the *Monkey Plugin* have a long maximal runtime. This is caused by *adb* connection problems that sometimes occur after a device reboot. The *Tripwire Plugin* needs a reboot into recovery mode to work, and the *Monkey Plugin* restarts the phone to trigger possi-

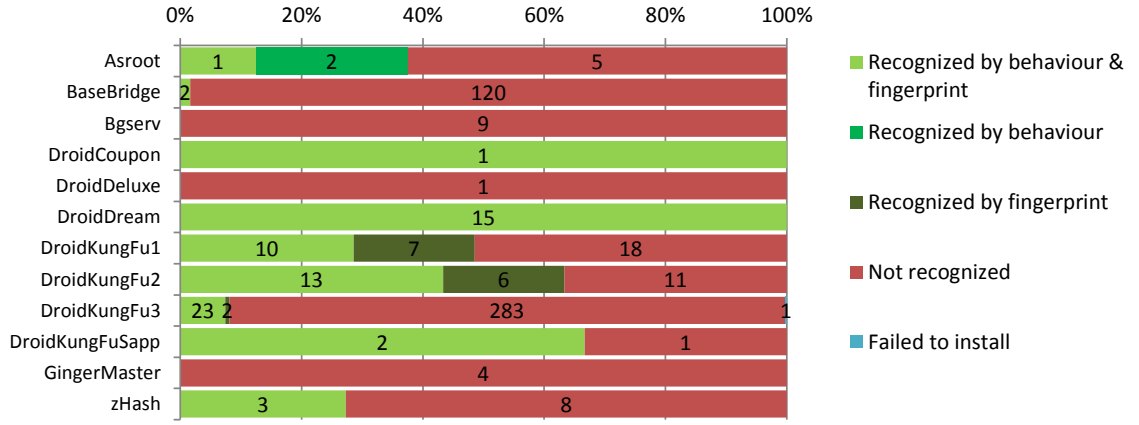

Figure 6.14: The improved detection rate of the *Tripwire Analysis* on a rooted phone with Internet access against the samples from the Malgenome project was 15.9%.

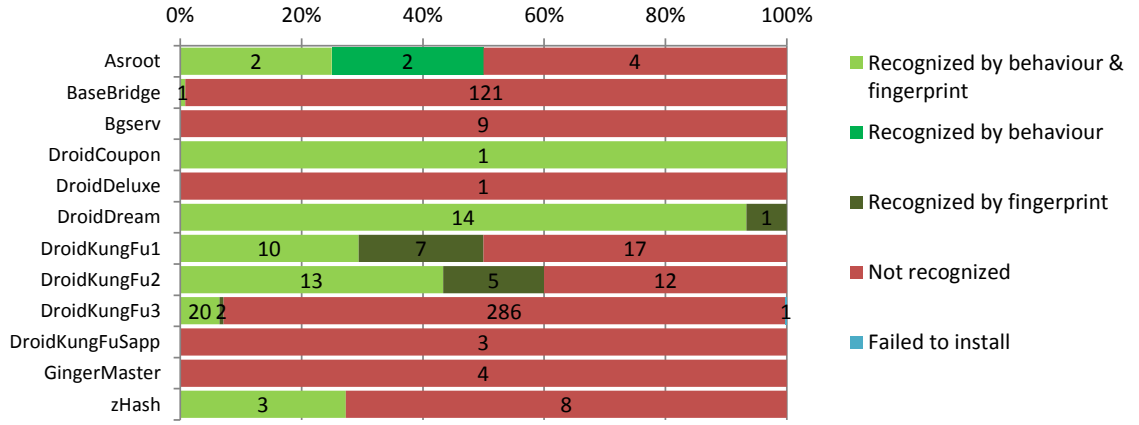

Figure 6.15: The improved detection rate of the *Tripwire Analysis* on a phone with a modified su-binary and Internet access against the samples from the Malgenome project was 14.8%.

ble *BOOT_COMPLETED*[24] broadcast receivers. The framework tries to reconnect, but from time to time a manual intervention is necessary (physically reconnect the phone or restart it). Figure 6.18 shows the runtime of the analysed applications. This figure shows that it took on average about thirteen minutes to analyse an application. The figure moreover shows that only during a few *Analyses* connection problems occurred that increased the *Analysis* time. The time-axis is cut at 50 minutes to increase clarity. The maximal runtime was over 5.5 hours (335 minutes).

[24] "android.intent.action.BOOT_COMPLETED"

Figure 6.16: The improved detection rate of the *Tripwire Analysis* on an unrooted phone with Internet access against the samples from the Malgenome project was 11.4%.

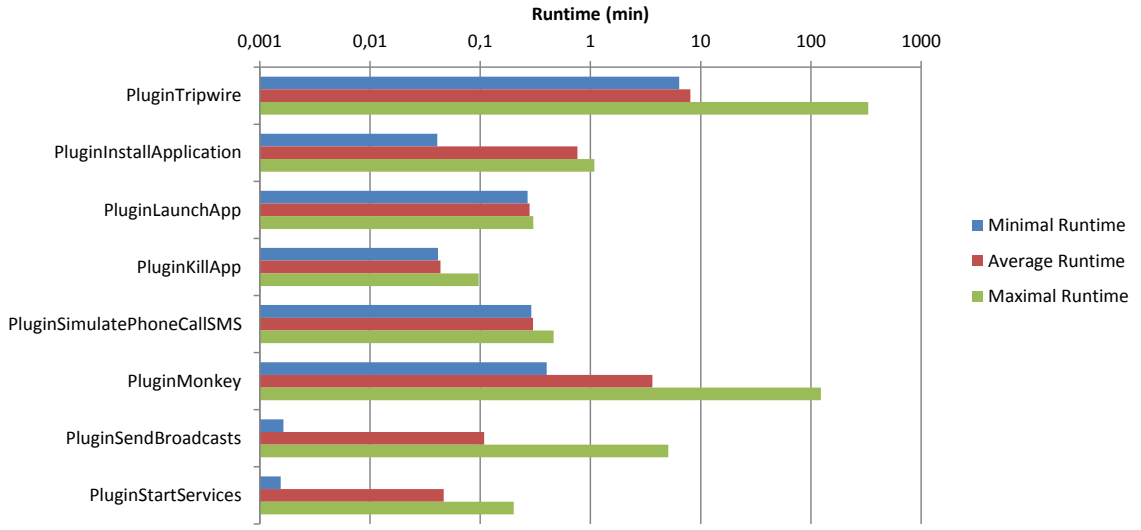

Figure 6.17: Runtime of *Plugins* executed during the *Tripwire Analysis*.

Figure 6.19 shows the *suspicious score* of all analysed applications. It shows that many analysed applications did not show a suspicious behaviour during the analysis. About half of the as suspicious marked applications have a low *suspicious score* of one or two and only a few applications behaved really suspicious.

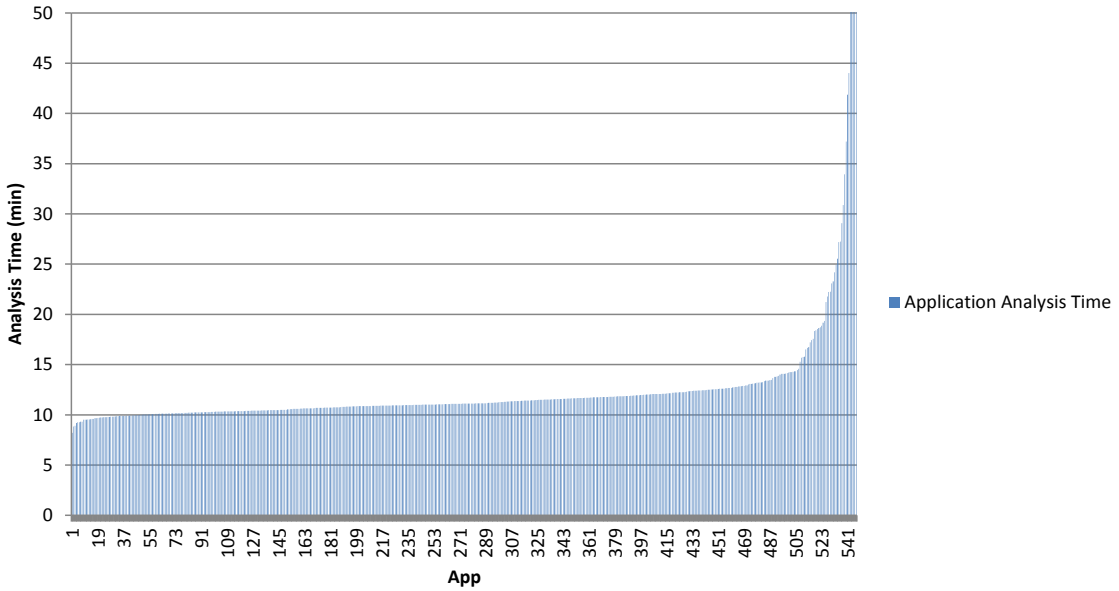

Figure 6.18: Application analysis time with *Tripwire Analysis*.

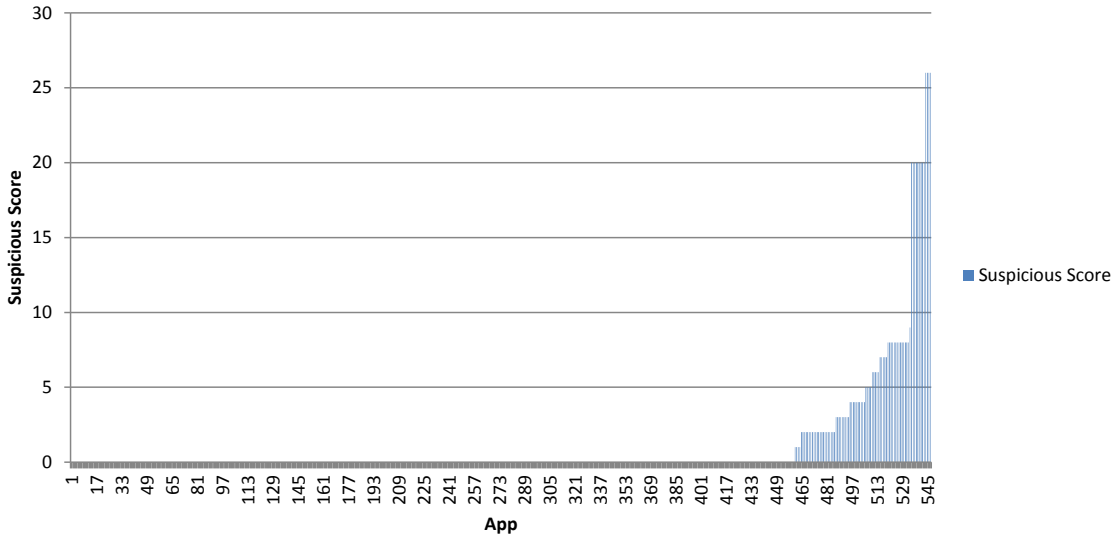

Figure 6.19: The calculated *suspicious score* of the analysed applications.

6.3.3 Non-Root Malware

The *Tripwire Analysis* was also used to analyse the remaining samples of the *Malgenome* project. On the one hand these samples can be used to evaluate the false positive rate, on the other hand for some families the literature is conflicting whether they use root exploits or not. This time the phone with the modified su binary has the highest detection rate, namely 6.0%, followed by the rooted phone with 5.1%, and the non-rooted phone reaches 0.14%. The run on the phone with the modified su binary has detected every sample that was detected by the two other runs. Therefore, the results will be discussed based on these findings. Figure 6.20 shows the detection rate. As expected, most families were not detected, because they do not use root exploits. The *DroidDreamLight* sample was marked as suspicious because it copied the APK of the NoLock application. This sample is a repacked version of an application backup program. Therefore, this sample can be seen as a false positive. Listing 6.5 shows the as suspicious marked files of a sample of the *DroidKungFu4* family. This sample added and replaced system binaries, which makes it suspicious. The other suspicious marked samples of the *DroidKungFu4* family show a similar behaviour. The two as suspicious marked samples of the *Geinimi* family and the 15 as suspicious marked samples of the *jSMSHider* family were marked as suspicious because they executed the modified su binary. As the system was not modified, these findings are probably false positives caused by the root usage of the benign part of the applications.

The *Tripwire Analysis* is suited to detect malware that use root exploits. The best results were achieved with the rooted phone. Besides the general drawback of dynamic analysis techniques, it seems that this *Analysis* is hampered the most by files “hidden” in the *assets* or *res* folder of the APK. While this drawback is only an issue when the malicious code is not executed, it could be fixed by using a static pre-check. This static pre-check could extract the contents of the APK and match it e.g. against the blacklist. A big advantage of this *Analysis* is that a physical device is used, which makes it very hard for malware to detect that it is analysed. This feature could be further improved if only the *User Plugin* is used. Also using the recovery mode for information collection seems to be a good approach. So far

```
added file:suspect=2, path=/sdcard/root.txt, modified=Fri Jan 04 03:50:28 CET ↩
    2013, size=91, permissions=-rwxrwxrwx, owner=root, group=root, hash=800b31526↩
    ceb57b9e6f280ebd46f495d7462a2a2
added file:suspect=2, path=/system/bin/dhcpcdd, modified=Fri Jan 04 02:47:17 CET ↩
    2013, size=44540, permissions=-rwxr-xr-x, owner=root, group=shell, hash=819b↩
    b001b402f8eec0e8ae87f69c50012ab985d6
added file:suspect=2, path=/system/bin/installdd, modified=Fri Jan 04 02:47:18 CE↩
    T 2013, size=18176, permissions=-rwxr-xr-x, owner=root, group=shell, hash=e4b↩
    ce349349ef9bcf6fdaa67731540f83ee61aaa
added file:suspect=2, path=/system/etc/.dhcpcd, modified=Fri Jan 04 02:47:10 CET ↩
    2013, size=18316, permissions=-rw-rw-rw-, owner=root, group=root, hash=5c7↩
    b198241c97179f20e00b966548f86d6c7d3f2
added file:suspect=2, path=/system/etc/.rild_cfg, modified=Fri Jan 04 02:47:12 CE↩
    T 2013, size=44, permissions=-rw-rw-rw-, owner=root, group=root, hash=00794ff↩
    df6c6174147f87b06e1c93190131567bf
added file:suspect=2, path=/system/xbin/ccb, modified=Fri Jan 04 02:47:11 CET ↩
    2013, size=18316, permissions=-rwsr-xr-x, owner=root, group=root, hash=5c7↩
    b198241c97179f20e00b966548f86d6c7d3f2
changed file was:suspect=1, path=/system/bin/dhcpcd, modified=Sun Dec 23 16:43:23↩
     CET 2012, size=44540, permissions=-rwxr-xr-x, owner=root, group=shell, has↩
    h=819bb001b402f8eec0e8ae87f69c50012ab985d6
changed file  is:suspect=2, path=/system/bin/dhcpcd, modified=Fri Jan 04 02:47:17↩
     CET 2013, size=18316, permissions=-rwxr-xr-x, owner=root, group=shell, has↩
    h=5c7b198241c97179f20e00b966548f86d6c7d3f2
changed file was:suspect=1, path=/system/bin/installd, modified=Sun Dec 23 ↩
    16:43:23 CET 2012, size=18176, permissions=-rwxr-xr-x, owner=root, group=shel↩
    l, hash=e4bce349349ef9bcf6fdaa67731540f83ee61aaa
changed file  is:suspect=2, path=/system/bin/installd, modified=Fri Jan 04 ↩
    02:47:18 CET 2013, size=18316, permissions=-rwxr-xr-x, owner=root, group=shel↩
    l, hash=5c7b198241c97179f20e00b966548f86d6c7d3f2
```

Listing 6.5: DroidKungFu4, True positive detected files

no malware is known that can hide itself in the recovery mode.

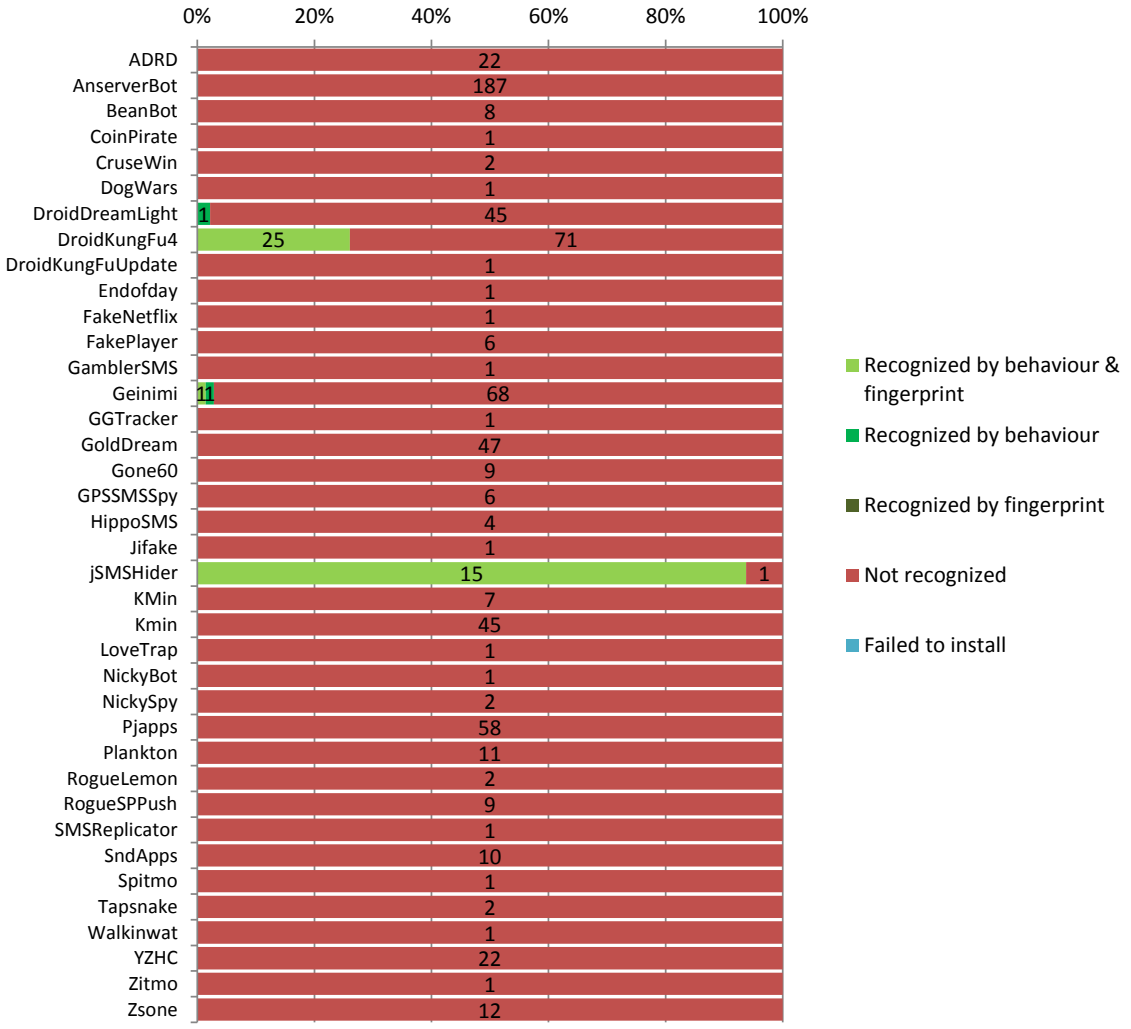

Figure 6.20: The detection rate of the *Tripwire Analysis* on a phone with a modified su-binary and Internet access against the "non-root" samples from the Malgenome project was 6.03%.

Chapter 7

Conclusion and Outlook

Android is the most prevalent mobile operating system. The popularity and the presence of sensitive data make it a lucrative target for malware authors.
This work presents a malware-detection framework for Android applications. The functionality of the framework can be extended by *Plugins*. The framework allows to create easily different *Analyses*. It provides methods that manage the communication with the Android device, which allows to control it or e.g. get the *logcat* output. To demonstrate the functions and the flexibility of the framework two *Analyses* and ten *Plugins* were developed. One *Analysis* is based on Tripwire. It compares a snapshot of the file system that is created prior the execution of the to be analysed application with a snapshot that is created after the execution and tries to find signs of root usage. This *Analysis* aims to detect malware that uses root exploits or achieved root privileges through other ways. The second *Analysis* is based on the DroidBox sandbox [19, 20]. This *Analysis* tries to detect malicious applications that leak sensitive data or produce costs for the user by sending short messages or starting phone calls.
The 1260 malware samples of the Malgenome project were analyst with these two *Analyses*. 510 from the 1260 samples were detected which leads to a detection rate of 39.8%. The detection rate can be explained by the drawbacks of dynamic analysis methods. Only one execution path is evaluated, which means if for some reason the malicious code is not executed the dynamic methods will not examine it. There can be several reasons why the malicious code is not executed. For example, during the

analysis several malware samples were found that tried to connect to a C&C server which was not available anymore and, therefore, the applications did nothing malicious. Another reason could be that the malware detected that it was analysed or executed in an emulator. Static analysis can help to detect such malware samples. Bergler [18] is currently working on *Analyses* and *Plugins* that use static analyses techniques. When these *Analyses* are finished and the results are merged with the results of the dynamic analyses the detection rate should improve.
We think combining static and dynamic analysis techniques is the best way to improve the detection rate. However, the detection rate can also be increased by improving the dynamic analyses. One way to improve the results of dynamic analyses could be to analyse the contents of the added and modified files. The *Tripwire Plugin* can copy these files to the host machine, what misses is a *Plugin* that evaluates the contents. Other problems of the dynamic analyses like the *adb* connection problem can probably be solved by using a newer Android version. Tripwire should work on newer Android versions, as long as a recovery mode is available that supports *adb* connections and some basic shell commands. The *DroidBox Plugin* should also work with newer Android versions, because in the meantime a new DroidBox image was released which supports Android 4.1.1. However, both dynamic *Analyses* could be sped up by restricting the execution time of the *Monkey Plugin* to about 10 minutes. The results showed that this can be done without losing relevant information.

Appendix A

Acronyms

adb Android Debug Bridge

AOSP Android Open Source Project

API Application Programming Interface

APK Application Package File

AVD Android Virtual Device

C&C Command & Control

IMEI International Mobile Equipment Identity

IMSI International Mobile Subscriber Identity

IPC Interprocess Communication

JNI Java Native Interface

PC Personal Computer

JSON JavaScript Object Notation

SMS Short Message Service

SVM Support Vector Machine

UID user identifier

VM virtual machine

Bibliography

[1] Android. Android. http://www.android.com/about/, 2012. Accessed: 08.11.2012.

[2] Android_Developers. https://plus.google.com/+AndroidDevelopers/posts/jHLD6HTfx9U, 2012. Accessed: 08.11.2012.

[3] Jon Fingas. Android tops 81 percent of smartphone market share in Q3. http://www.engadget.com/2013/10/31/strategy-analytics-q3-2013-phone-share/, 2013. Accessed: 08.01.2014.

[4] Android Developers. Android, the world's most popular mobile platform. http://developer.android.com/about/index.html, 2012. Accessed: 13.11.2012.

[5] Android. Android Security Overview. http://source.android.com/tech/security/, 2012. Accessed: 13.11.2012.

[6] Android Developers. Platform Versions. http://developer.android.com/about/dashboards/index.html, 2012. Accessed: 13.11.2012.

[7] Emil Protalinski. New Android malware infects 100,000 Chinese smartphones. http://www.zdnet.com/new-android-malware-infects-100000-chinese-smartphones-7000000497/, 2012. Accessed: 7.11.2012.

[8] Rick Merritt. More than a third of Android apps host malware. http://cdn.eetimes.com/electronics-news/4391305/More-than-a-third-of-Android-apps-host-malware, 2012. Accessed: 7.1.2013.

[9] Hans Jörg Maron. Trend Micro warnt vor Android-Malware-Flut . http://www.inside-it.ch/articles/29457, 2012. Accessed: 7.11.2012.

[10] Emil Protalinski. Android malware numbers explode to 25,000 in June 2012. http://www.zdnet.com/android-malware-numbers-explode-to-25000-in-june-2012-7000001046/, 2012. Accessed: 7.11.2012.

[11] Ted Samson. Slow patching puts Android users at further risk. http://www.infoworld.com/t/mobile-security/slow-patching-puts-android-users-further-risk-198668, 2012.

[12] Dan Rampe. Android Malware Increased by 3,325% Last Year. http://www.threatmetrix.com/fraudsandends/uncategorized/android-malware-increased-by-3325-last-year/, 2012.

[13] Rowena Diocton. Android Malware: How Worried Should You Be? http://blog.trendmicro.com/trendlabs-security-intelligence/android-malware-how-worried-should-you-be/, 2012.

[14] Jürgen Schmidt and Achim Barczok. FAQ: Android und Sicherheit. http://www.heise.de/ct/hotline/FAQ-Android-und-Sicherheit-1647138.html, 2012.

[15] Kaspersky Lab Global Research and Analysis Team. Kaspersky Security Bulletin 2013. 2013. URL http://media.kaspersky.com/pdf/KSB_2013_EN.pdf.

[16] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4681-0. doi: 10.1109/SP.2012.16. URL http://dx.doi.org/10.1109/SP.2012.16.

[17] Mahinthan Chandramohan and Hee Beng Kuan Tan. Detection of Mobile Malware in the Wild. *Computer*, pages 1–14, 2012. ISSN 0018-9162. doi: 10.1109/MC.2012.36. URL http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6133256.

[18] Bernd Bergler. Android malware detection framework based on static analysis. Master's thesis, Graz, Technical University, 2014.

[19] Patrik Lantz. DroidBox. http://code.google.com/p/droidbox/, 2011.

[20] Patrik Lantz. An Android Application Sandbox for Dynamic Analysis. 2011.

[21] GH Kim. The design and implementation of tripwire: A file system integrity checker. *Proceedings of the 2nd ACM Conference on*, 1994. URL http://dl.acm.org/citation.cfm?id=191183.

[22] Ben Elgin. Google Buys Android for Its Mobile Arsenal. http://www.businessweek.com/stories/2005-08-16/google-buys-android-for-its-mobile-arsenal, 2005.

[23] AVG PR. AVG unveils globalCommunity Powered Threat Report - Q3-2011. http://now.avg.com/avg-unveils-globalcommunity-powered-threat-report-q3-2011/, 2011. Accessed: 02.03.2012.

[24] Mattias via XDIN Android Blog. The Android boot process from power on. http://www.androidenea.com/2009/06/android-boot-process-from-power-on.html. Accessed: 02.08.2013.

[25] Florian Schmidt. Bootprozess. http://www.droidwiki.de/Bootprozess, 2013. Accessed: 02.08.2013.

[26] Ali Waqas. Bootloader. http://www.addictivetips.com/mobile/what-is-bootloader-and-how-to-unlock-bootloader-on-android-phones-complete-guide/, 2010. Accessed: 02.08.2013.

[27] Oracle and/or its affiliates. jarsigner. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html. Accessed: 02.08.2013.

[28] Android Developers. Application Fundamentals. http://developer.android.com/guide/components/fundamentals.html, 2012. Accessed: 13.11.2012.

[29] Android Developers. Services. http://developer.android.com/guide/components/services.html, 2012. Accessed: 13.11.2012.

[30] Android Developers. Content Providers. http://developer.android.com/guide/topics/providers/content-providers.html, 2012. Accessed: 13.11.2012.

[31] Android Developers. Activities. http://developer.android.com/guide/components/activities.html, 2012. Accessed: 13.11.2012.

[32] Android Developers. Intents and Intent Filters. http://developer.android.com/guide/components/intents-filters.html, 2013. Accessed: 02.08.2013.

[33] Android Developers. Android Permissions. http://developer.android.com/reference/android/Manifest.permission.html, 2012. Accessed: 13.11.2012.

[34] Android. Android Security Overview. http://source.android.com/devices/tech/security/#the-android-permission-model-accessing-protected-apis, 2013. Accessed: 02.08.2013.

[35] Clemens Orthacker, Peter Teufl, Stefan Kraxberger, Günther Lackner, Michael Gissing, Alexander Marsalek, Johannes Leibetseder, and Oliver Prevenhueber. Android Security Permissions - Can We Trust Them? In *Security and Privacy in Mobile Information and Communication Systems*, volume 94 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 40–51. Springer Berlin Heidelberg, 2012. URL http://dx.doi.org/10.1007/978-3-642-30244-2_4.

[36] Hiroshi Lockheimer. Android and Security. http://googlemobile.blogspot.co.at/2012/02/android-and-security.html, 2012.

[37] Jon Oberheide. Dissecting Android's Bouncer. https://blog.duosecurity.com/2012/06/dissecting-androids-bouncer/, 2012.

[38] Rich Cannings. Exercising Our Remote Application Removal Feature. http://android-developers.blogspot.co.at/2010/06/exercising-our-remote-application.html, 2010.

[39] Fahmida Y. Rashid. Android's Biggest Security Threat: OS Fragmentation. http://securitywatch.pcmag.com/android/308966-android-s-biggest-security-threat-os-fragmentation. Accessed: 08.04.2013.

[40] Jamie Lendino. Why Android Fragmentation Is Still a Problem. http://www.pcmag.com/article2/0,2817,2406991,00.asp, 2012. Accessed: 08.11.2012.

[41] Michael Grace, Yajin Zhou, Zhi Wang, Xuxian Jiang, and Oval Drive. Systematic Detection of Capability Leaks in Stock Android Smartphones. *North*, 2012. URL http://handysmarkt.com/gehen/http://www.csc.ncsu.edu/faculty/jiang/pubs/NDSS12_WOODPECKER.pdf.

[42] Artem Russakovskii. Massive Security Vulnerability In HTC Android Devices (EVO 3D, 4G, Thunderbolt, Others) Exposes Phone Numbers, GPS, SMS, Emails Addresses, Much More. http://www.androidpolice.com/2011/10/01/massive-security-vulnerability-in-htc-android-devices-evo-3d-4g-thunderbolt-others-exposes-phone-numbers-gps-sms-emails-addresses-much-more/, 2011.

[43] AP Felt, Matthew Finifter, Erika Chin, and Steven Hanna. A survey of mobile malware in the wild. *smartphones and mobile*, 2011. URL http://dl.acm.org/citation.cfm?id=2046618.

[44] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *ACSAC*, pages 421–430. IEEE Computer Society, 2007. URL http://dblp.uni-trier.de/db/conf/acsac/acsac2007.html#MoserKK07.

[45] V. Benjamin Livshits and Monica S. Lam. Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, pages 18–18, Berkeley, CA, USA, 2005. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1251398.1251416.

[46] N. Ayewah, D. Hovemeyer, J.D. Morgenthaler, J. Penix, and William Pugh. Using static analysis to find bugs. *Software, IEEE*, 25(5):22–29, Sept 2008. ISSN 0740-7459. doi: 10.1109/MS.2008.130.

[47] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few

billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM*, 53(2):66–75, February 2010. ISSN 0001-0782. doi: 10.1145/1646353.1646374. URL http://doi.acm.org/10.1145/1646353.1646374.

[48] P. Louridas. Static code analysis. *Software, IEEE*, 23(4):58–61, July 2006. ISSN 0740-7459. doi: 10.1109/MS.2006.114.

[49] C. Artho and A Biere. Applying static analysis to large-scale, multi-threaded java programs. In *Software Engineering Conference, 2001. Proceedings. 2001 Australian*, pages 68–75, 2001. doi: 10.1109/ASWEC.2001.948499.

[50] A-D. Schmidt, R. Bye, H.-G. Schmidt, J. Clausen, O. Kiraz, K.A Yuksel, S.A Camtepe, and S. Albayrak. Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC '09. IEEE International Conference on*, pages 1–5, June 2009. doi: 10.1109/ICC.2009.5199486.

[51] Alexandre Bartel, Jacques Klein, Yves Le Traon, and Martin Monperrus. Dexpler: Converting android dalvik bytecode to jimple for static analysis with soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, pages 27–38, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1490-9. doi: 10.1145/2259051.2259056. URL http://doi.acm.org/10.1145/2259051.2259056.

[52] A Shabtai, Y. Fledel, and Y. Elovici. Automated static code analysis for classifying android applications using machine learning. In *Computational Intelligence and Security (CIS), 2010 International Conference on*, pages 329–333, Dec 2010. doi: 10.1109/CIS.2010.77.

[53] Étienne Payet and Fausto Spoto. Static analysis of android programs. *Information and Software Technology*, 54(11):1192 – 1201, 2012. ISSN 0950-5849. doi: http://dx.doi.org/10.1016/j.infsof.2012.05.003. URL http://www.sciencedirect.com/science/article/pii/S0950584912001012.

[54] T. Bläsing, L. Batyuk, A.-D. Schmidt, S.A. Camtepe, and S. Albayrak. An Android Application Sandbox System for Suspicious Software Detection. *Tech-*

niques, pages 55–62, 2010. doi: 10.1109/. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5665792.

[55] Golam Sarwar, Olivier Mehani, Roksana Boreli, and Mohamed Ali Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In Pierangela Samarati, editor, *SECRYPT*, pages 461–468. SciTePress, 2013. ISBN 978-989-8565-73-0. URL http://dblp.uni-trier.de/db/conf/secrypt/secrypt2013.html#SarwarMBK13.

[56] William Enck, Landon P Cox, Peter Gilbert, and Patrick Mcdaniel. TaintDroid : An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. *Design*, pages 1–6, 2010. URL http://appanalysis.org/tdroid10.pdf.

[57] Yinfeng Qiu. Bypassing Android Permissions: What You Need to Know. http://blog.trendmicro.com/trendlabs-security-intelligence/bypassing-android-permissions-what-you-need-to-know/, 2012.

[58] Yeongung Park, ChoongHyun Lee, Chanhee Lee, JiHyeog Lim, Sangchul Han, Minkyu Park, and Seong-Je Cho. Rgbdroid: A novel response-based approach to android privilege escalation attacks. In *Presented as part of the 5th USENIX Workshop on Large-Scale Exploits and Emergent Threats*, Berkeley, CA, 2012. USENIX. URL https://www.usenix.org/conference/leet12/rgbdroid-novel-approach-effective-response-privilege-escalation-attacks-android.

[59] Juanru Li, Dawu Gu, and Yuhao Luo. Android malware forensics: Reconstruction of malicious events. In *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 552 –558, june 2012. doi: 10.1109/ICDCSW.2012.33.

[60] Xuetao Wei, Lorenzo Gomez, and Iulian Neamtiu. Malicious Android Applications in the Enterprise: What Do They Do and How Do We Fix It? *ICDE Workshop on Secure*, 2012. URL http://www.cs.ucr.edu/~neamtiu/pubs/sdmsm12wei.pdf.

[61] Aubrey-Derrick Schmidt, Jan Hendrik Clausen, Seyit Ahmet Camtepe, and Sahin Albayrak. Detecting symbian os malware through static function call analysis. In *Proceedings of the 4th IEEE International Conference on Malicious and Unwanted Software (Malware 2009)*, pages 15–22. IEEE, 2009. doi: 10.1109/MALWARE.2009.5403024.

[62] Manuel Egele, Christopher Kruegel, and Engin Kirda. PiOS : Detecting Privacy Leaks in iOS Applications. *Sophia*, 2011. URL `https://www.seclab.tuwien.ac.at/papers/egele-ndss11.pdf`.

[63] William Enck, Damien Octeau, and P McDaniel. A study of android application security. *the 20th USENIX security*, 2011. URL `http://www.usenix.org/event/sec11/tech/slides/enck.pdf`.

[64] Axelle Apvrille and Tim Strazzere. Reducing the window of opportunity for Android malware Gotta catch 'em all. *Journal in Computer Virology*, 8(1-2):61–71, apr 2012. ISSN 1772-9890. doi: 10.1007/s11416-012-0162-3. URL `http://www.springerlink.com/index/10.1007/s11416-012-0162-3`.

[65] Bryan Dixon and S Mishra. On Rootkit and Malware Detection in Smartphones. *Current*, pages 162–163, 2010. URL `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5542600`.

[66] Min Zhao, Tao Zhang, Fangbin Ge, and Zhijian Yuan. RobotDroid: A Lightweight Malware Detection Framework On Smartphones. *Journal of Networks*, 7(4):715–722, apr 2012. ISSN 1796-2056. doi: 10.4304/jnw.7.4.715-722. URL `http://ojs.academypublisher.com/index.php/jnw/article/view/6527`.

[67] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "Andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 2011. ISSN 09259902. doi: 10.1007/s10844-010-0148-x. URL `http://www.springerlink.com/index/10.1007/s10844-010-0148-x`.

[68] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. *of the 16th ACM conference on*, pages 235–245, 2009. URL http://dl.acm.org/citation.cfm?id=1653691.

[69] Georgios Portokalidis and Herbert Bos. Paranoid Android : Versatile Protection For Smartphones. *Network Security*, pages 347–356, 2008.

[70] Iker Burguera and Urko Zurutuza. Crowdroid : Behavior-Based Malware Detection System for Android. *Science*, 2011. URL http://dl.acm.org/citation.cfm?id=2046619.

[71] Hahnsang Kim, Joshua Smith, and Kang G. Shin. Detecting energy-greedy anomalies and mobile malware variants. In *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, pages 239–252, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-139-2. doi: 10.1145/1378600.1378627. URL http://doi.acm.org/10.1145/1378600.1378627.

[72] Lei Liu, Guanhua Yan, Xinwen Zhang, and Songqing Chen. Virusmeter: Preventing your cellphone from spies. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID '09, pages 244–264, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-04341-3. doi: 10.1007/978-3-642-04342-0_13. URL http://dx.doi.org/10.1007/978-3-642-04342-0_13.

[73] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. *of the 19th Annual Network and*, 2012. URL http://www.csd.uoc.gr/~hy558/papers/mal_apps.pdf.

[74] Google. Google Play. https://play.google.com/store, 2012. Accessed: 13.11.2012.

[75] Martin Kropp and Pamela Morales. Automated gui testing on the android platform. *on Testing Software and Systems: Short Papers*, page 67, 2010.

[76] Stefan Piotrowski. Automated Testing for Android. 2011.

[77] Android Developers. UI/Application Exerciser Monkey. http://developer.android.com/tools/help/monkey.html, 2012. Accessed: 13.11.2012.

[78] Android Developers. Android Debug Bridge. http://developer.android.com/tools/help/adb.html, 2012. Accessed: 13.11.2012.

[79] Lena Tenenboim-Chekina, Oren Barad, Asaf Shabtai, Dudu Mimran, Lior Rokach, Bracha Shapira, and Yuval Elovici. Detecting application update attack on mobile devices through network features. In *INFOCOM*, 2013.

[80] Xuxian Jiang. An Evaluation of the Application ("App") Verification Service in Android 4.2. http://www.cs.ncsu.edu/faculty/jiang/appverify/, 2012. Accessed: 08.01.2013.

[81] Mario Ballano and Takashi Katsuki. Android.Adrd. http://www.symantec.com/security_response/writeup.jsp?docid=2011-021514-4954-99&tabid=2, 2011.

[82] Alexey Podrezov. Trojan: Android/BaseBridge.A. https://www.f-secure.com/v-descs/trojan_android_basebridge.shtml, 2011.

[83] Piotr Krysiuk Stephen Doherty. Android.Basebridge. http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99&tabid=2, 2011.

[84] Dave Smith. Android: Are raw resources stored locally on the filesystem? http://stackoverflow.com/questions/7421001/android-are-raw-resources-stored-locally-on-the-filesystem, 2011.

[85] Xuxian Jiang. Security Alert: New BeanBot SMS Trojan Discovered. http://www.csc.ncsu.edu/faculty/jiang/BeanBot/, 2011.

[86] Mark Balanza. Android Malware Acts as an SMS Relay. http://blog.trendmicro.com/trendlabs-security-intelligence/android-malware-acts-as-an-sms-relay/, 2011.

[87] Xuxian Jiang. Security Alert: New DroidKungFu Variant – AGAIN! – Found in Alternative Android Markets. http://www.csc.ncsu.edu/faculty/jiang/DroidKungFu3/, 2011.

[88] Tim Wyatt. Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild. https://blog.lookout.com/blog/2010/12/29/geinimi_trojan/, 2010.

[89] OGorman Gavin and Honda Hatsuho. Android.Geinimi. http://www.symantec.com/security_response/writeup.jsp?docid=2011-010111-5403-99&tabid=2, 2011.

[90] Microsoft. Trojan:AndroidOS/Kmin.A. http://www.microsoft.com/security/portal/threat/encyclopedia/entry.aspx?Name=Trojan:AndroidOS/Kmin.A#tab=2, 2011.

[91] Xuxian Jiang. Security Alert: New Android SMS Trojan – YZHCSMS – Found in Official Android Market and Alternative Markets. http://www.csc.ncsu.edu/faculty/jiang/YZHCSMS/, 2011.

[92] Roee Hay. Android SQLite Journal Information Disclosure - CVE-2011-3901. pages 1–5, 2012.