Hochschule Darmstadt

-Faculty of Computer Science-

Porting and Improving an Android Sandbox for Automated Assessment of Malware

A thesis submitted in fulfillment of the requirements
for the degree *Master of Science (M.Sc.)*

by
Christoph Dietzel, B.Sc.

Examiner:	*Prof. Dr. Harald Baier*
Second Examiner:	*Prof. Dr. Alois Schütte*
Supervisor:	*Dr. Michael Spreitzenbarth*

Emission Date:	02.10.2013
Submission Date:	07.04.2014

Declaration of Authorship

I, *Christoph Dietzel*, declare that this thesis titled, 'Porting and Improving an Android Sandbox for Automated Assessment of Malware' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Darmstadt, 07.04.2014 _____

Abstract {English}

Smartphones are an integral part of our daily routine. With being the most widely used mobile operating system, cyber criminals naturally extended their malicious activities towards *Android*. Security analysts recognized an alarming increase in *Android* malware families of 390% from 2012 to 2013.

The major challenge in analyzing this massive amount of malware samples is a growing number of employed obfuscation techniques disguising the malicious portions of source code from analysts. Sandboxes are able to overcome obfuscation by executing malware within an isolated environment. Unfortunately, sophisticated malware can determine that it is running within an analysis environment and dynamically adapt its behavior to be considered as benign.

To conquer this limitation, we extend the *Android* application sandbox *DroidBox* to be more resilient towards detection techniques and additionally feature compatibility with up-to-date *Android* applications. Thus, this Thesis is divided into two, not directly related, challenges: First we verified the accurate operation of *DroidBox 4.1* and utilized it as a base for our continuative porting to the most recent version of *Android*. Thereby we semi-automated the porting procedure to aid further developments. Second, we investigate defense strategies applied by *Android* malware to thwart dynamic analysis. A taxonomy is developed and leveraged to cluster a huge amount of practically applicable sandbox evasion techniques. Finally, we propose anti-detection measures in alignment with the taxonomy and successfully tackle all introduced evasion techniques. Consequently, from malware's point of view our extension of *DroidBox* is indistinguishable from a real device.

We demonstrate our detection methods to not only be effective against all existing online sandboxes, but also putting the defenders a step ahead by assisting analysts in combating evasive mobile malware through an improved version of *DroidBox*. Ultimately, it is integrated into the online analysis services *Mobile-Sandbox*.

Abstract {German}

Smartphones sind mittlerweile ein fester Bestandteil unseres täglichen Lebens. *Android* ist dabei die am weitesten verbreitete Plattform und deshalb auch zunehmend das Ziel von Computerkriminalität. Von 2012 auf 2013 konstatieren Sicherheitsexperten ein alarmierendes Wachstum von *Android* Schadsoftware-Familien um 390%.

Die Herausforderung bei der Analyse dieser großen Mengen von Schadsoftware ist der steigende Einsatz von Verschleierungstechniken, die den bösartigen Programmcode vor Analytikern verbergen. Sandboxen sind in der Lage diese zu überwinden und führen die Schadprogramme in einer isolierten Umgebung aus. Leider können hochentwickelte Computerviren feststellen, dass sie in einer Analyseumgebung ausgeführt werden und adaptieren ihr Verhalten dynamisch, um gutartig zu erscheinen.

Um diese Einschränkung zu eliminieren, wurde *DroidBox*, eine Sandbox für *Android*, widerstandsfähiger gegen solche Detektionsmaßnahmen gemacht. Außerdem wird die Kompatibilität zu neusten *Android* Anwendungen hergestellt. Somit gliedert sich diese Arbeit in zwei nicht unmittelbar themenverwandte Abschnitte: Zunächst verifizieren wir die korrekte Funktionsweise von *Droidbox 4.1* und verwenden es als Basis für unsere anschließende Portierung auf die neueste Version von *Android*. Dabei automatisieren wir diesen Prozess teilweise, um spätere Portierungsvorhaben zu unterstützen. Zweitens untersuchen wir die von *Android* Schädlingen eingesetzten Sandboxdetektionsmaßnahmen. Hierfür wird ein Klassifikationsverfahren entwickelt, mit dessen Hilfe sich eine Vielzahl von praktischen Erkennungsmethoden zusammenfassen lässt. Schließlich zeigen wir Antidetektionsmaßnahmen, mit der wir alle zuvor entwickelten Erkennungsstrategien erfolgreich abwehren können. Aus Sicht von Schadsoftware kann die erweiterte *DroidBox* nicht von einem echten Gerät unterschieden werden.

Es wird gezeigt, dass unsere Erkennungsmethoden nicht nur wirksam gegen alle bestehenden Online-Sandboxen sind, sondern auch, dass die Verbesserung der *DroidBox* Analysten wirksam bei der Bekämpfung von Computerviren unterstützt. Abschließend wurde die gehärtete *DroidBox* in die *Mobile-Sandbox* Umgebung integriert.

Acknowledgements

I would like to thank all the people who contributed in some way to the creation of this Thesis. Foremost, I thank my supervisor Michael Spreitzenbarth for the opportunity to work at such an interesting cutting-edge research topic, his patience, and the always valuable feedback.

Special thanks to the Siemens AG and Siemens CERT in particular for funding my research and thereby the work described in this Master's Thesis.

I also thank the staff and members of the Faculty of Computer Science and the Biometrics and Internet-Security Research Group at Hochschule Darmstadt for the productive and always friendly environment to study in the past years. Especially the projects and seminars were an enrichment thanks to the personal commitment of Sebastian Abt and Harald Baier.

Special thanks to the reviewers of this Thesis for the important suggestions and the helpful feedback, particularly Sven Ossenbühl and Michael Ziemann.

Most importantly, I want to thank my parents, friends, and girlfriend for their support during my study time and for always encouraging me when times got hard.

Contents

List of Figures

List of Tables

Listings

Abbreviations

3G	**3**(Third) **G**eneration
ADB	**A**ndroid **D**ebug **B**ridge
API	**A**pplication **P**rogramming **I**nterface
ARM	**A**dvanced **R**ISC **M**achine
ART	**A**ndroid **R**un**T**ime
AVD	**A**ndroid **V**irtual **D**evice
AV	**A**nti **V**irus
CPU	**C**entral **P**rocessing **U**nit
DVM	**D**alvik **V**irtual **M**achine
GPS	**G**lobal **P**ositioning **S**ystem
HTML	**H**yper **T**ext **M**arkup **L**anguage
IDE	**I**ntegrated **D**evelopment **E**nvironment
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
IMEI	**I**nternational **M**obile **E**quipment **I**dentity
IMSI	**I**nernational **M**obile **S**ubscriber **I**dentity
IPC	**I**nter **P**rocess **C**ommunication
IP	**I**nternet **P**rotocol
ITU	**I**nternational **T**elecommunications **U**nion
IT	**I**nformation **T**echnology
JIT	**J**ust **I**n **T**ime
JNI	**J**ava **N**ative **I**nterface
JVM	**J**ava **V**irtual **M**achine
L1	**L**evel **1** (Cache)

OS	**O**perating **S**ystem
PC	**P**ersonal **C**omputer
POSIX	**P**ortable **O**perating **S**ystem **I**nterface
SDK	**S**oftware **D**evelopment **K**it
SHA	**S**ecure **H**ash **A**lgorithm
SIM	**S**ubscriber **I**dentity **M**odule
SMS	**S**hort **M**essage **S**ervice
SQL	**S**tructured **Q**uery **L**anguage
SoC	**S**ystem **o**n a **C**hip
TCP	**T**ransmission **C**ontrol **P**rotocol
URL	**U**niform **R**esource **L**ocator
VMM	**V**irtual **M**achine **M**onitor
VM	**V**irtual **M**achine
WLAN	**W**ireless **L**ocal **A**rea **N**etwork
XML	E**x**tensible **M**arkup **L**anguage

Don't complain. Just work harder.
And be prepared. Luck is truly where preparation meets opportunity.

Randy Pausch

Chapter 1

Introduction

1.1 Motivation

Over the last decades information technology (IT) became an indispensable part of our everyday lives. Particularly the interconnectedness of IT systems via the Internet has reached unprecedented levels. According to the International Telecommunications Union (ITU), 2.75 billion individuals are using the Internet which equals 38.8% of the world population [36]. Developed countries have, as expected, significantly more Internet users: USA 81%, Germany 84%, Spain 72%, or Netherlands 93% [87]. While inhabitants of the mentioned countries embraced the convenience of the Internet into their daily routines, the developing countries are expected to push the number of users within the next decades even more.
Initially IT supported traditional businesses in terms of increasing efficiency and performance, but meanwhile it rather transformed into an independent business branch. The online-based economy including shipped goods had a turnover of 39.8 billion euros in Germany [11] and 384.8 billion dollars in the USA [19] in 2013. Aside from the total numbers the upturn of the e-commerce market is still continuing, as depicted in Figure 1.1.

The aforementioned monetary valuation of the Internet as well as the sheer number of users do not remain without consequences: Criminals are attracted by the new market. Whereas hackers were merely persuaded by fun in the early days, cyber criminals determined the opportunity to gain financial profit from this new cyber world [34, 89]. The size of the global market of cyber crime was estimated to exceed one trillion dollars

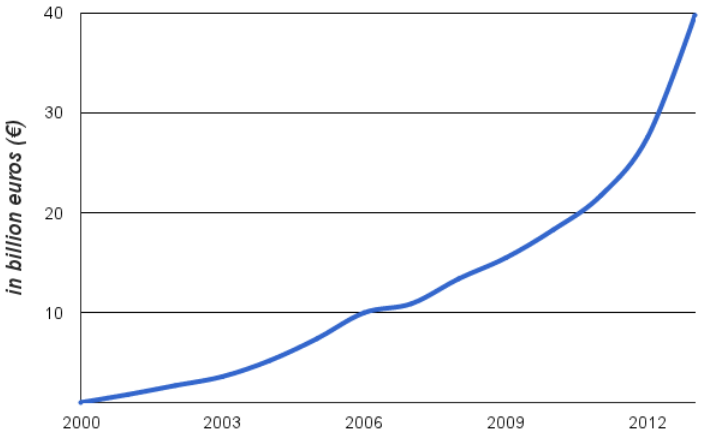


FIGURE 1.1: E-commerce sales growth in Germany [82].

according to the Vice-President of the European Commission responsible for the Digital Agenda of Internet Security [41]. Criminals attempt to gain control over computer systems or networks in order to achieve their financial goals. Therefore the attackers typically exploit vulnerabilities which unfortunately are enclosed in any software or hardware. Programs that infect systems through such loopholes, referred to as malware, may cause harm to users (cf. Section 2.2). Consecutively, we illustrate the threat of cyber crime by revealing numbers and facts published by IT security vendors:

- For a survey 3,716 Internet users from selected countries were asked if they have ever been infected by malware. Although users might be unaware of an infection, 58% admitted a malware infestation [37].
- Kaspersky counted 5,188,740,554 cyber attacks on user's computer systems in 2013 [38].
- Symantec counted on average 160 targeted attacks per day in 2013, which is an increase of 42% compared to 2012. The number of exposed identities per breach rose to 604,826 [86].
- In 2012 74,000 unique malicious web domains were registered [86].
- The absolute number of vulnerabilities detected in 2012 was 9,776, discovered in 2,503 products from 421 different vendors [74].
- Malware targets on stealing information, wiping data, blocking infrastructure operation, and stealing money [38].

As outlined by the given statistics, attackers have great impact and threaten our cyber space. Whereby in the beginning of the digital age the conventional way of accessing the Internet was to use ordinary computers, this is rapidly changing. We are shifting from a Internet society to a mobile Internet society [6]. Despite the high availability of wireless local area network (WLAN) access points, true mobility can only be provided through the cellular infrastructure. Thus, mobile providers extended the range and bandwidth of their infrastructure extensively within the last years. As of 2012 the population coverage of Third Generation (3G) mobile networks is comparatively high: USA 93%, Germany 90%, Spain 98%, Netherlands 99%, and Poland 69% [57]. As stated by Evans [21], Chief Futurist of Cisco, forecasts predict that over 50 Exabytes of data will be exchanged by mobile devices in 2015.

This massive bandwidth usage is caused by a steady rising number of *smart* devices. At the beginning of the new millennium a new device class emerged at the market: *Smartphones*. This revolution is driven by a profound change of mobile phone's assets. Smartphones have particularly advanced hardware and software capabilities. Typical hardware features are large screens (3 to 5 inch), touch input interfaces, reasonable computing power, several connectivity interfaces, and a diversity of sensors. These components are utilized by software running on a complete operating system (OS) similar to ordinary desktop systems. This allows the development of complex applications (referred to as apps) like full featured browsers, e-mail clients, office tools, or media players that increase the usability considerably. In fact, innovative new applications are published which make use of the mobility combined with device sensors such as geolocation (GPS) or acceleration sensors. Moreover, the tightly integrated apps can be installed at a given time of the user's choice through a distribution platform for third-party apps: The so called *app stores*. Not only in the private sector third-party apps play a key role. Enterprise software vendors such as SAP or Oracle supply apps to access business management systems on the go via mobile devices [64, 71].

Whereas we exemplified the rise of cyber crime along the ongoing growth of the Internet beforehand, we provide insight in how the described transformation towards mobile Internet usage indeed let attackers expand their vicious activities to aim at smartphones likewise. Hence, the higher goal can clearly be outlined as gaining financial profit, the question of direct incentives is not answered yet. Given the implication that the value of personnel information is greater than ever before, criminals are particularly interested in looting private data. As highlighted by Felt et al. [25], it is the most common target. Approximately 60% of the instances in their sample database collecting sensitive user

information. Address books may contain up to several hundred data sets and could be used for spam spreading or identity theft. Also credentials which are typically saved within apps to access services, such as social media or cloud storage services, running the risk of being purloined for account abuse. This turns into an even more drastic exposure if login information for bank accounts or payment details are concerned.
Besides, a common attack purpose is to generate revenue for attackers by the stealthy engagement of premium services [25, 80]. Despite the legitimation of premium-rate phone calls and short message service (SMS) messages for value added services, they are repeatedly instrumented by criminals. Costs for the users up to several euros per minute or message are easily generated. Furthermore, it could feasibly go unfolded until the user's next bank statement.
All the mentioned attack vectors lead to a situation where malicious software for mobile devices cannot be considered as an individual case anymore. Malware for the Symbian mobile platform has been around since a decade. With the dropping market share and the decreasing number of spotted malware families in the wild it becomes neglectable. In contrast, at the same time the increasingly popular mobile environment *Android* (cf. Section 2.1) turns into an profitable target, as illustrated in Figure 1.2. *FakePlayer*,

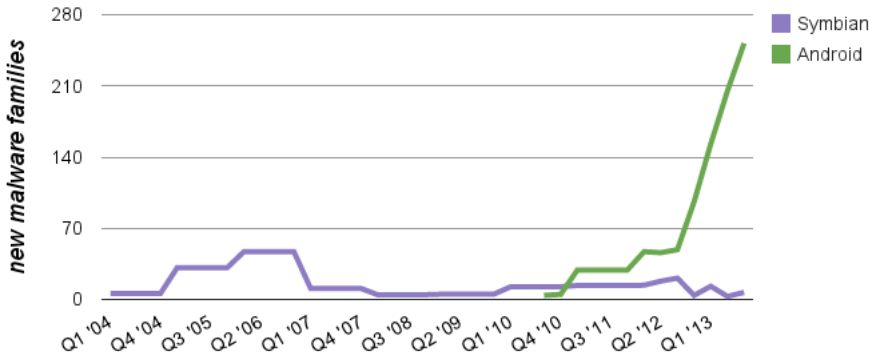


FIGURE 1.2: New malware families by quarter 2004 - 2013 [22–24].

the very first malware targeting Android, appeared in August 2010 [47]. Over time, however, we observed a continuous growth in the numbers of malicious Android applications. In early 2012 Zhou and Jiang presented their seminal findings on characterization and evaluation of Android malware wherein they discuss the need for better detection measures and anti malware solutions for mobile devices, due to the exploding growth of infection risk [101]. Recently released threat reports of security vendors exemplify the great amount of malicious applications which can possibly harm smartphones. Symantec recognized an increase about 58% of mobile malware families from 2011 to 2012 [86]. For 2013, F-Secure even stated a growth of 390% [23, 24] compared to 2012 as depicted

in Figure 1.2. In absolute terms, an average number of 550,000 new samples per quarter in the first three quarters of 2013 are counted by Intel Security (formerly McAfee) (cf. Figure 1.3) [50]. Hence, it is difficult to define what should be considered as a unique instance of a malicious app. In analogy to ordinary malware, authors try to evade detection by repackaging, recompiling, or other transformations of already identified samples. Obviously, the security vendors are interested in an preferably threatening analysis of the situation, since their business is to sell security solutions. As shown by Maggi et al. [46], there are already several products for Android mobile devices available.

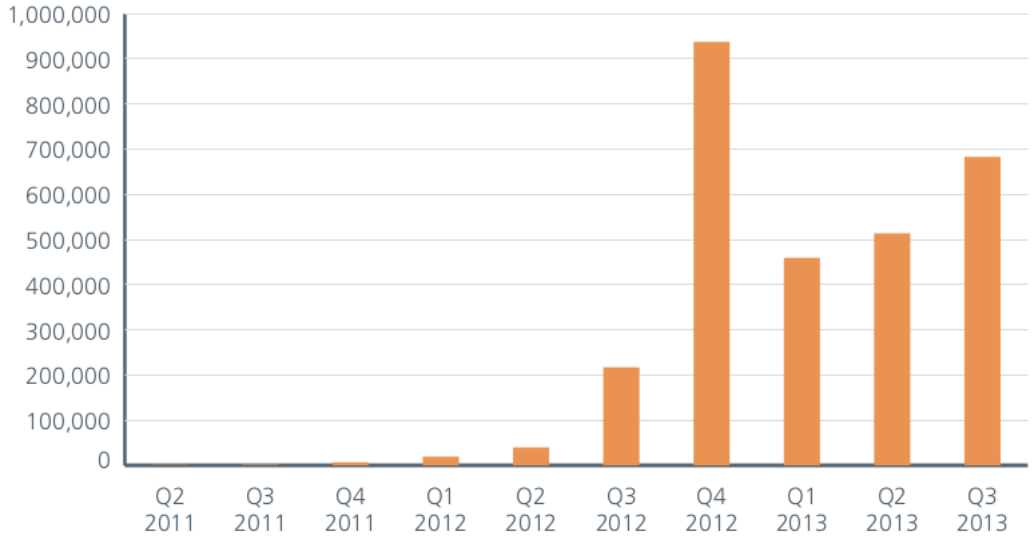


FIGURE 1.3: New unique malware samples by quarter 2011 - 2013 [50].

Nevertheless, the quintessence of both, scientific researchers and security vendors, clearly states the ongoing rise of mobile malware. In particular, the Android environment is targeted. With an estimated market share of about 80% of global smartphone sales (cf. Figure 1.4) it is the market leader even though it was just released in 2008. Google's Senior Vice President Sundar Pichai announced 1 billion device activations in September 2013. Beyond the device figures, Google's Play Store, the app distribution channel for Android, serves more than one million apps as of mid-2013 [96].

The open design of the Android platform allows users to install apps from unofficial sources like third-party stores. Indeed, neither the Play Store nor other stores, notwithstanding all their efforts, prevent distribution of malicious apps with absolute certainty. Scammers will find ways to circumvent protection measures to sneak malware into the stores. Especially Russian and Chinese platforms seem to be affected [49].

Due to the rampant growth, more applications must be analyzed in a given time to enable researchers, store operators, and security vendors to develop effective countermeasures.

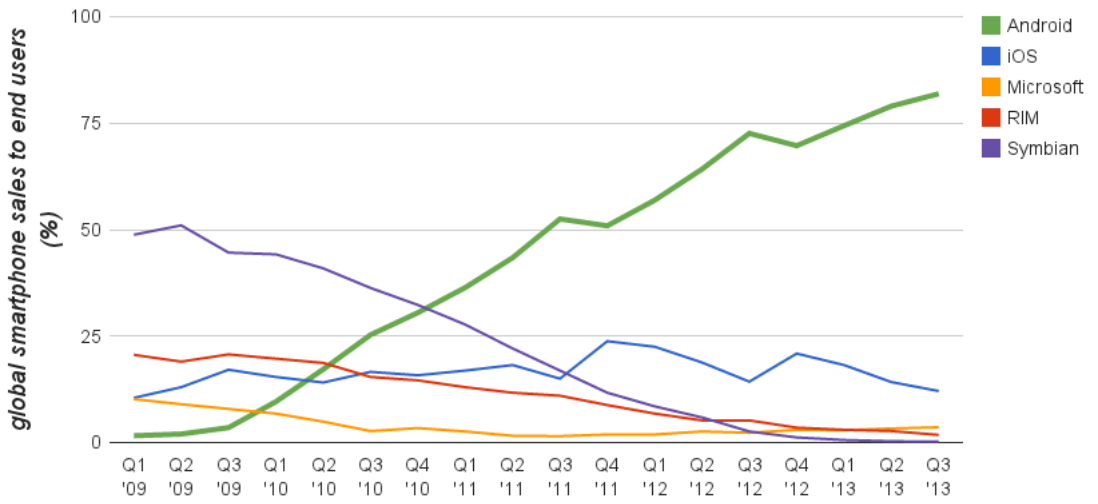


FIGURE 1.4: Global smartphone sales share to end users by quarter 2009 - 2013 [27, 83].

The traditional method which investigates each sample manually is error-prone and time consuming (cf. Section 2.3). Thus, it is required to use automated analysis techniques. In order to overcome code obfuscation techniques, which can prevent an effective static analysis, a large body of dynamic analysis research has been proposed [9, 20, 42, 81]. The so called sandboxes examine potential malware during execution in a controlled environment (cf. Section 2.3.2). Google established Bouncer, a service to scan submitted apps for malicious behavior, to limit the number of unwanted apps in their store [44]. However, research has shown that it is in fact possible to bypass it [56]. Attackers obviously aim on protecting their *"intellectual property"* and do not perform any malicious activity as soon as they detect that it is running within an analysis environment.

In conclusion, not only the total number of malware targeting Android is still growing, also its sophistication is steadily increasing. Moreover, the concepts practically applied for automated analysis environments lack effectiveness due to the explained evasion measures implemented by malware authors. We predict a similar development as seen in the ordinary computer sphere. In the near future we expect all serious Android malware families to at least employ strong obfuscation techniques or even sandbox evasion procedures. This calls for a continuously improvement of existing solutions. We stress that it is crucial to understand presently conducted sandbox detection strategies and to apply findings to existing solutions. The overarching goal is to analyze arbitrary applications in dynamic environments at any time and to conduct meaningful results even for upcoming generations of malware.

1.2 Problem Definition and Objectives

As already mentioned in Section 1.1, the market for smart mobile devices grew considerably and is believed to continue its rise in the near future. The Android platform has a significant market share and the number of available apps grows fast. Beyond the fact that it also appeals to criminals, they become even better organized. An increasing number of malware, and in particular more sophisticated ones are the consequence.
Thus, it is required to use automated analysis techniques in order to identify potential harmful software, to understand its behavior, and to develop adequate countermeasures. DroidBox is an open source sandbox to analyze Android applications. Unfortunately, it comes with an eminent drawback: Apps which are deployed for novel versions of Android cannot be examined. In fact, most applications deployed for outdated Android versions still run on up-to-date systems. In contrast, those which make use of the new features or changed APIs of later versions are not compatible anymore.
To address the changing market shares of the different versions, the existing version of the Android 2.3 based DroidBox sandbox has to be ported to version 4.x. The current version's shortcoming, that only a subset of the existing applications can be analyzed, would be eliminated. The porting is supported by an already existing version of TaintDroid, compatible with Android 4.1. TaintDroid is a taint tracking (cf. Section 2.3.3) system which is the basis for DroidBox [20]. Afterwards, the profound changes require an conscientious evaluation of the updated environment to prove the correctness.
In practice, regarding the continuous development of Android with its continuous major releases, such a port is required frequently. In order to alleviate this costly procedure, a preferably generic approach should be derived and outlined.

As aforesaid, sophisticated Android malware already employs defense strategies like the sandbox detection. If an instance is running within an analysis environment it could try to thwart it by changing the behavior and not revealing malicious actions. The applied evasion capabilities need to be examined and structured. Based on the gained insight, solutions to prevent a detection should be developed and implemented for the ported DroidBox. In order to demonstrate the effectiveness it will be integrated into a real world solution. This means, it is tested with a representable amount of malware samples. Finally, we give a comprehensive comparison of the features with other well-known online analysis platforms.

The briefly summarized objective is to build an Android sandbox which is notably harder to identify by malware in comparison to DroidBox. Additionally, it is supposed to be

compatible with up-to-date Android apps.

1.3 Out of Scope

Whereat we delineated the objectives for this Thesis in the previous chapter, a definition to clarify the work out of scope is given here.
Neither do we cover techniques to obfuscate source code, nor approaches to circumvent such obfuscation techniques.
Furthermore, this work does not contribute to increase the robustness of taint tracking by utilizing static information flow analysis. Therefore we refer to "Detecting Control Flow in Smartphones", published by Graa et al. in 2012 [30]. Nevertheless, insight on the limitations is given in Section 2.3.3.
Although evasion tactics and anti detection for Android are addressed in depth, however, measures exploiting so-called *rootkits* to circumvent detection are not. We stress, once malware resides deeply in the kernel it is infeasible either to inspect its behavior or to prevent it from detecting the sandbox. For related reading please refer to *"A Guide to Kernel Exploitation"* by Perla and Oldani [66].
We are also not covering advancements of emulation software in order to mitigate sandbox detection.
When the practical work for this thesis had already been completed, TaintDroid 4.3 was released. Hence we do not take it into account.

1.4 Thesis Outline

In order to follow the ideas of the authors, the reader is encouraged to read all chapters of this Thesis sequentially in their given order. The structure of this Thesis is as follows:

Chapter 1 gives an introduction to the current threats of the Internet and the risk of malware infections for mobile devices. Thereby it presents the motivation and objectives for this work.

Chapter 2 presents the required background for the Android platform (Section 2.1). Dynamic analysis systems in general (Section 2.3.2), and the Droidbox sandbox (Sections 2.3.4 & 2.3.5) are explained as a concrete implementation which is extended within the context of this Thesis.

Chapter 3 outlines the procedure of porting DroidBox to a more recent version of Android. First, the developed version 4.1 of DroidBox is evaluated (Section 3.2). At second, DroidBox is ported to the latest version of Android (Section 3.3). Moreover, the approach to automate porting of prospective versions is discussed.

Chapter 4 introduces a taxonomy to cluster dynamic analysis detection methods. In alignment with it, a huge amount of techniques to unveil sandboxes is given (Section 4.2). Terminally, a complete list of measures to counter the previously described detection vectors is demonstrated and implemented (Section 4.3).

Chapter 5 evaluates the proposed sandbox detection techniques and countermeasures. Therefore, a set of online sandboxes is examined and compared to the improved sandbox environment (Section 5.1). In addition, the integration of the extended DroidBox into the Mobile Sandbox is briefly illustrated (Section 5.2).

Chapter 6 discusses the results and contributions (Section 6.1). Ultimately, an outlook on possible future work is given subsequently (Section 6.2).

Chapter 2

Foundations

After introducing the motivation and objectives of this Thesis in Chapter 1, we cover the essential foundations required to understand the work that is explained later on in Chapters 4 and 5.
In the beginning we provide some details about the mobile operating system Android, whereby we discuss the layered system architecture. Followed by an introduction of mobile malware in Section 2.2 and an explanation of how it can be analyzed in Section 2.3. Thereafter, we compare the two analysis approaches and explain the sandbox concept. Since the main part of this work describes the improvement of DroidBox, the necessary insight into DroidBox (Section 2.3.5) and its base TaintDroid (Section 2.3.4) is given. Finally, we will summarize the essence of this chapter.

2.1 Android

This section provides insights into the Android OS. Android is a *Linux* based open source OS for mobile devices with Advanced RISC Machine (ARM) architecture [2], developed by the Open Handset Alliance under the leadership of Google. Its market share continues to grow rapidly as already outlined in Chapter 1.

The first publicly available Android version has been released in September 2008. All consecutive major releases are listed in Table 2.1 with the corresponding release dates. The latest Android OS version is 4.4 alias KitKat which was released in October 2013. Any released version came with notable changes, some regarding the design others with

Version	Codename	Release Date
1.0	Base	September 2008
1.5	Cupcake	April 2009
1.6	Donut	September 2009
2.0	Eclair	October 2009
2.2	Froyo	May 2010
2.3	Gingerbread	December 2010
3.x	Honeycomb	February 2011
4.0	Ice Cream Sandwich	October 2011
4.1	Jelly Bean	June 2012
4.2	Jelly Bean	November 2012
4.3	Jelly Bean	July 2013
4.4	KitKat	October 2013

TABLE 2.1: The history of Android major releases.

massive modifications at the system's architecture. For a more detailed explanation please refer to the release notes [62].

The Android software stack contains four layers, as illustrated in Figure 2.1: *Linux kernel*, *libraries/runtime environments*, *application framework*, and *application layer*. They are connected by interfaces to assure a robust inter-layer communication. The following discusses the different components (cf. Figure 2.1) [31, 76].

Google instruments a modified version of the open source *Linux kernel* as base for Android. It comes with a few special additions such as wakelocks, a memory management system, and security mechanisms. Most changes are due to the fact that system resources are limited on mobile embedded platforms.
The next upper layer contains *system libraries* and the *runtime environments*, mostly written in C/C++. The libraries are modules of code that are compiled down to native machine code, comparable to those commonly used by other Linux distributions. All libraries are required by the system itself and by third-party applications likewise.
Android possesses two different runtime environments. The *Dalvik Virtual Machine* (*DalvikVM* or *DVM*) is the default runtime environment and executes applications and system services, each in its own virtual machine (VM) process to provide a sandboxed environment. In Android 4.4, the *Android Runtime* (*ART*) has been introduced experimentally and will eventually replace the DVM in future releases [61]. Its main goal is to increase execution performance by precompiling applications at installation and not at runtime.
Despite the strict separation of the VMs there is a communication channel for exchanging

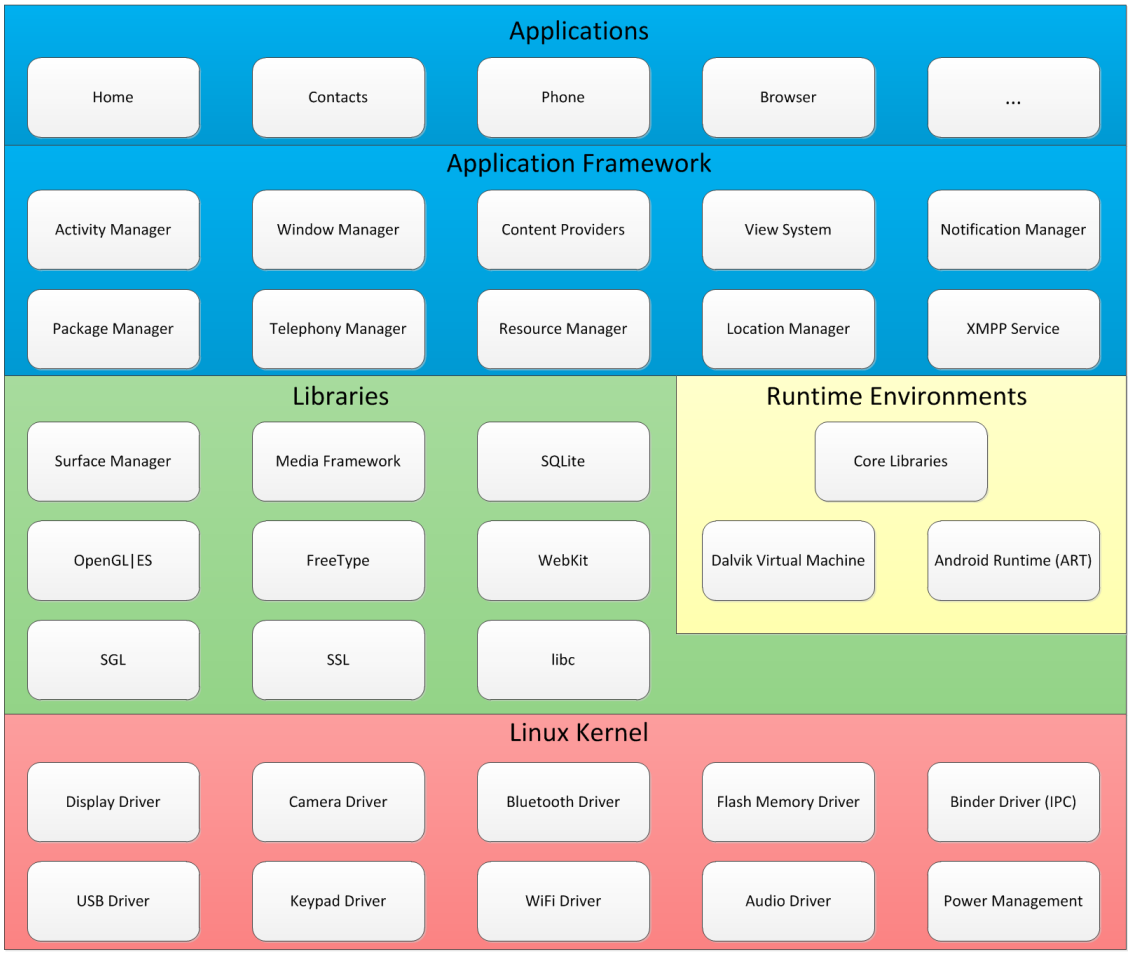


FIGURE 2.1: The Android operating system software stack with its four layers [1].

information cross process boundaries. This inter-process communication (IPC) mechanism is called *Binder*, which allows high level components to interact with Android's system services in a simple manner.

The *application framework* layer exposes system functionality through an easy-to-use application programming interface (API). The services run as a background process and are divided into modular components depending on their functionality such as Search Service, Telephony Manager, or Window Manager (cf. Figure 2.1).

The uppermost layer is the *application* layer. All third-party and some preinstalled applications, like Contacts or Browser, running inside the DVM on this layer.

The primary programming language for applications is Java, whereat the source code is compiled into a machine code-like bytecode (.class). It can be interpreted by the Java Virtual Machine (JVM), a computer architecture independent runtime environment. For Android, however, Google introduced the Dalvik bytecode (.dex) which is derived from JVM bytecode and executable by the DalvikVM. The major difference is that JVM bytecode is stored in one or more files and loaded dynamically at runtime,

while Dalvik bytecode (even of multiple classes) is stored in one file only. As stated in Figure 2.2, class files are transformed by the dx tool[1] into a single dex file. Along with all its resources and a manifest it is packed into an Android package file (.apk) which can be distributed and installed on Android devices. The manifest file describes, among other general information, the capabilities of its application. The compile process, as implemented by modern integrated development environments (IDE) like Eclipse, is represented by the green arrows in Figure 2.2. Typically, that is not made transparent to the developer. An explanation of the reverse process and the smali file format is given in the next section.

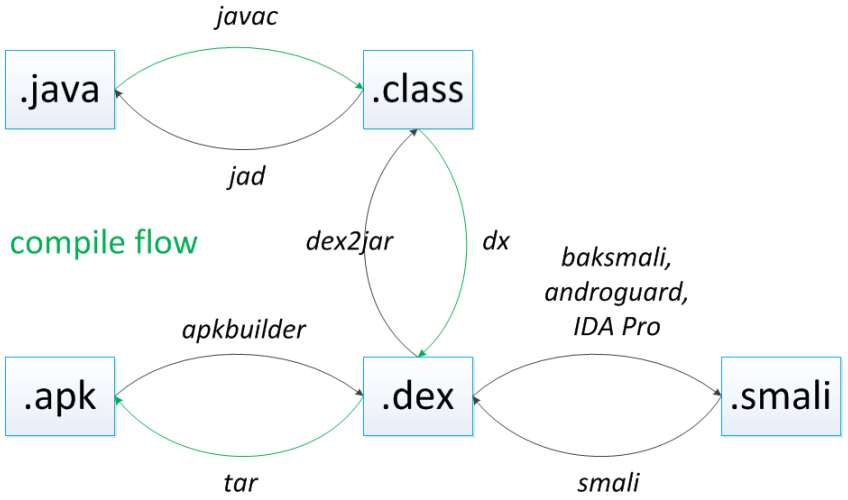


FIGURE 2.2: The Android compile flow and its tools.

A key feature for the success of Android is the distribution model. Third-party applications are conveniently accessible through app store platforms. Even developers benefit since they do not need to be concerned of the distribution chain or the handling of payment. As of July 2013, Google's app store (Play Store) served one million apps [96]. Besides, several unofficial app stores exist (e.g. Amazon App-Shop, Yandex.Store, 1Mobile, etc.).

A shortcoming of the Android update philosophy is the failure of providing security patches in a reasonable time frame. Updates are only delivered with new releases and not as a continuous process for bug fixing. Considering the release cycle, a critical number of devices is exposed to attackers due to unpatched security holes, not even taking into account the time required by the smartphone manufacturers to adapt the new OS version to their devices. Table 2.2 presents an overview of the version's distribution as of January 2014 [63].

[1] The **dx tool** is part of the Android software development kit.

Codename	**API vers.**	**Share**
Froyo	8	1.3%
Gingerbread	10	21.2%
Honeycomb	13	0.1%
Ice Cream Sandwich	15	16.9%
Jelly Bean	16, 17, 18	59.1%
KitKat	19	1.4%

TABLE 2.2: The Android version distribution [63].

2.2 Mobile Malware

Malware, short for **mal**icious soft**ware**, terms software which *"deliberately fulfills the harmful intent of an attacker"* [52]. To allow an instant correlation to a specific kind of harmful behavior, malware is commonly classified with terms such as *virus*, *worm*, or *trojan*. Malicious software exists for all computer platforms but originates form Personal Computers (PC). Traditionally, cyber criminals were driven by earning respect or demonstrating security vulnerabilities. Nowadays financial gain is the main motivation [34, 89, 102].

As pointed out in Chapter 1, the number of Android malware increased significantly within the last years and the growth does not seem to slow down any soon. This trend has been predicted by Becher et al. [6]. They identified the reasons as follows: First, the total number of smartphones grew rapidly since they became more powerful and cheaper, transmission capabilities of wireless networks grew while the prices decreased at the same time, and they are easily third-party extensible as we already outlined in the last section. Hence, a single piece of malware is capable of infecting far more devices. Second, smartphones contain more sensitive information simultaneously, such as payment details, authentication credentials, personal data (identity theft), or business sensitive information compared to ordinary computers.

Mobile devices are exposed to several fundamental different attack vectors. These threats are classified according to their target as follows [6, 79]:

- *Hardware-centric attacks* target physical vulnerabilities and are not exploitable remotely.
- Attacks aiming at the communication infrastructure or back-end systems belong to *device-independent attacks*.
- *Software-centric attacks* are attacks against software running on a mobile device, including system apps, the OS itself, and drivers. In contrast to Becher [6], we

include also so-called social engineering attacks into this class. These attacks evade technical security measures by leveraging the trust of users who are not aware of IT security foundations.

Software-centric attacks demand an opportunity to let target systems execute a portion of code (the malware) on behalf of the attacker. Thus, it calls for methods to infect devices (*infection vectors*) outlined below.
Android malware commonly utilizes *repackaging* attacks [101]. In essence, popular applications are downloaded and decompiled by the attackers, the malware's payload is added, the app is recompiled, and uploaded to an app store. Users may confuse these repackaged versions with the legitimated benign ones and install them onto their device. Since the harmful code is an integral part of the application, this could expose its presence to security software, either on the device or inside the app store already. Instead of injecting the payload before distribution, malware authors make use of the update facilities of most app stores referred to as *update attack* [101]. The base versions do not contain malicious code and are therefore considerably harder to detect. At a time of the attackers choice an update is performed and the harmful content is fetched.
Also, malware authors tend to simply name their software to sound alike popular ones, but do not have anything else in common. The described infection vectors exploit one of the three properties:

- Android allows its users to install software from untrusted sources, e.g. third-party stores. Those stores are famous especially in Russia and Asia [4, 49, 86].
- The malware analyses performed by the stores are not sophisticated enough [56].
- The user's decision-making process, prior installing a new app, whether it is malicious or benign is not supported enough by the OS and by context information given by the stores.

Furthermore, traditional exploitation of software vulnerabilities in order to infect Android devices begins to rise. Modern software is generally composed of thousands or millions lines of code. Hence, it is very unlikely that it does not contain any bugs. Malware exploits those bugs to break into systems, sophisticated pieces are even able, in contrast to social engineering attacks, to do so without the user's interaction. However, to date such attacks are very rare but will probably emerge more often in the future.

Finally, we give an overview of threat classification subsequently, according to Felt et al. [25] and the refinement of Spreitzenbarth [79].

- **Malware** is used by attackers to obtain control over a mobile device. The purpose is either to steal data or to perform criminal actions over the foreign smartphone. In rare cases malware caused damage to the device. The possible infection path are vulnerabilities or social engineering attacks, as denoted above.

- **Spyware** collects information about the device's owner and sends it to the person who put it in place. According to Felt, spyware is commonly installed by an attacker who has physical access [25].

- **Greyware**, in contrast to malware and spyware, has legitimate functionality which conceals its true goals. Therefore users install it unintentionally onto their devices. Nevertheless, it collects far more information than needed for the service offered and is often driven by marketing purposes. Those apps do not get banned from the Play Store in every case since they do not violate the terms and conditions.

- **Fraudware**'s main ambition is to use premium services for making profit. The advertised functionality is available after the user has sent premium Short Message Service (SMS) messages. Indeed the user gets informed about the costs even though the authors try to hide the reference about upcoming charges.

Within this section we provided an overview of mobile malware in general and Android malware in specific. The sophistication of threats is rising and attacks are relevant in practice. On that account the next section gives insight in the analysis of malware.

2.3 Malware Analysis

To successfully protect users and infrastructures from malware threats two conditions must be met. First, suspicious applications need to be analyzed and classified as either malicious or benign. Second, a piece of software once determined as malware needs to be detected at any system without a repeated analysis.
Since the detection strategy of security software bases mainly upon "some sort of signature matching process to identify known threats" [18], each new sample has to be distinguished. Thereby another drawback has to be challenged: Signatures are easily changeable through obfuscation techniques such as encryption or packaging [75]. However, each sample has to be classified as a variation of an already known threat or a yet unknown one which requires further manual analysis. A sufficient proceeding is to automate malware analysis. This can either be done *statically* or *dynamically* [8]. *Static*

analysis involves examining the code of an executable without running it, while a *dynamic* analysis is performed by running the sample in a safe environment [75]. Both approaches are described and compared in Section 2.3.1.

As illustrated in the introduction (cf. Chapter 1), the fast growing quantity of malware, besides the increase in professionalism, is the main challenge for security vendors and researchers. It appears that a classical manual analysis of suspicious applications is infeasible under this circumstances, since this process is error-prone and time-consuming. Therefore, both techniques, static and dynamic should be automated up to a certain degree to support the analyst's decision whether an additional manual analysis is reasonable or not.

2.3.1 Static vs. Dynamic

The classical approach for automated malware analysis is called *static* analysis. It is concerned with the examination of source code or meta information obtained by disassembling or decompiling of a malware sample.
As aforementioned, Android application packages (.apk) can be transformed back into Java code by reversing the compile process (cf. Figure 2.2). The Android bytecode files (.dex) are extracted from the app's archive (.apk). As the next step to restore the initial source code, the dex file is converted back into traditional Java bytecode (.class). The dex2jar tool accomplishes this step and allows the commitment of graphical Java decompilers (e.g. jad), which finally delivers human readable code. Unfortunately, the transformation is error-prone and hence the result is not satisfactory in every case. Thus it is required in some cases to transform the Android bytecode into a new language named smali (.smali). Smali is rather simple to understand in comparison to bytecode but not as convenient as Java. In order to demonstrated the differences Listing 2.2 exemplifies a portion of smali code while Listing 2.1 shows the original Java source of a "Hello World" sample application.

```
public class MainActivity extends Activity {

  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    sayHello("Hello World!", 3);
  }

  private void sayHello(String text, int duration){
```

```
        Toast.makeText(this, text, duration).show();
    }
}
```

LISTING 2.1: Example of Java source code.

```
[...]
.method private sayHello(Ljava/lang/String;I)V
    .registers 4
    .param p1, "text"    # Ljava/lang/String;
    .param p2, "duration"    # I
    .prologue
    .line 17
    invoke-static {p0, p1, p2}, Landroid/widget/Toast;->makeText
      Landroid/content/Context;Ljava/lang/CharSequence;I)Landroid/widget/Toast;
    move-result-object v0
    invoke-virtual {v0}, Landroid/widget/Toast;->show()V
    .line 18
    return-void
.end method

.method protected onCreate(Landroid/os/Bundle;)V
    .registers 4
    .param p1, "savedInstanceState"    # Landroid/os/Bundle;
    .prologue
    .line 11
    invoke-super {p0, p1}, Landroid/app/Activity;->onCreate(Landroid/os/Bundle;)V
    .line 12
    const/high16 v0, 0x7f030000
    invoke-virtual {p0, v0}, L[...]/MainActivity;->setContentView(I)V
    .line 13
    const-string v0, "Hello World!"
    const/4 v1, 0x3
    invoke-direct {p0, v0, v1}, L[...]/MainActivity;->sayHello
      (Ljava/lang/String;I)V
    .line 14
    return-void
.end method
```

LISTING 2.2: The reversed smali code of Listing 2.1.

The major advantage of analyzing static code is that it can be performed fast while it is relatively simple at the same time [53]. On the other hand malware employs obfuscation techniques which aims to conceal the source code from the analyst [98]. Thus, the code is unreadable or dynamically reloaded at an arbitrary point of time during execution and is not available for the static analysis. An Android specific obfuscation approach is to hide malicious code blocks outside the Dalvik runtime. For instance, in native

system libraries or app sections which are written in the C programming language and accessible through the Java Native Interface (JNI).

In contrast, *dynamic* analysis does not investigate code but rather refers to techniques which execute malware samples within a safe, isolated environment. It allows an investigation without putting the own productive system or network at risk. Such *sandboxes* (cf. Section 2.3.2) use specially prepared operating systems to monitor the applications behavior and its interaction with other software. After a defined period of time a report is generated automatically. It contains details including called API methods, modified data at the hard drive, performed encryption, and network traffic.
Dynamic analysis is able to combat obfuscation techniques. Since it analyzes the sample at runtime, access to the state of the system (calls, variables) is assured at any time. Furthermore, several tracking systems manage to track variables while they propagate through the systems (cf. Section 2.3.3). Certainly, authors of malware do not comply with analysis environments which circumvent their deployed obfuscation measures. Thence, they began to implement runtime detection methods to determine if the application is running inside a sandbox [69]. In Chapter 4 we provide an overview of Android detection schemes and suggest effective countermeasures.

In order to overcome the limitations of static and dynamic analysis it makes sense to combine both techniques. Such a combined implementation can be found, among others, in the Mobile-Sandbox [51, 79]. The next section describes the composition of an analysis environment for Android.

2.3.2 Sandbox

Sandboxes feature isolated systems for the execution of untrusted or malignant code within a realistic environment to conduct dynamic analyzes. Whereby isolation denotes an important property for securing the analysis system, it restricts the access to critical resources like communication interfaces or hardware devices. Furthermore, operations performed by the malware can be observed to provide a better understanding of its behavior. In spite of that, sandboxes are designed in a manner to not be unveiled. The sample under analysis is supposed to behave exactly as it would on a real device. In this sense, sandboxes are virtualized operating systems with specific additions.
In the domain of conventional PCs powerful sandboxes emerged already more than a decade ago, e.g. CWSandbox [99], Anubis [5], or Java sandboxes [29]. However, the development lacked of Android enabled systems until recently. In October 2010,

Blaessing et al. were the first to present a sandbox for Android [9]. At the same time Enck et al. presented TaintDroid [20] (cf. Section 2.3.4), which has been augmented subsequently by DroidBox [35] (cf. Section 2.3.5), TraceDroid [92], SandDroid [10], Mobile-Sandbox [79, 81], and Andrubis [93].

The Android emulator as part of the Software Development Kit (SDK) is the base for all enumerated sandboxes. Its main purpose is to support efficient testing for developers. Therefore it uses the open source QEMU virtualization solution which supports full system emulation for the ARM architecture [7]. Additional devices like display, audio, internal and external storage, network interfaces, and generic devices can be simulated as well. Android device's core hardware component is the System on a Chip (SoC) that is fully emulated and called *Goldfish*. The Android Virtual Device (AVD) manager provides a wrapper to define the concrete Android system's configuration which is loaded into QEMU. Ultimately, the emulator can run the Android OS within an emulated environment without being distinguished by an ordinary app. It allows to trigger certain system events as phone calls, incoming SMS, or geo locations from outside the emulator.

In order to leverage the Android emulator as sandbox for dynamic malware analysis it needs to be embedded into a framework which enables an automated injection of suspicious apps into the system, the monitoring for system events of interest, and a feature to collect and report the obtained information. Such systems have been enumerated before already. Since the Taint Tracking approach plays a crucial role for all those systems, it is introduced in the next subsections.

2.3.3 Taint Tracking

Taint tracking, also known as taint checking, information flow tracking, or data tainting, is a technique which allows to track flows of information while propagating through a computer system, in order to increase the system's security. Therefore a set of critical information sources is defined. Besides, all critical processing units within the system are defined as so-called *sinks*. The definition of sources or a sinks in practice highly depends on the environment's context. If the aim is to keep track of string propagation, for instance to prohibit command injection attacks within a web application, all user inputs would be sources. All methods that execute assigned code would be sinks. In contrast, an environment which is designed to keep track of sensitive information within a system would define methods which retrieve the information as sources. Interfaces to other systems, such as network interfaces, would be defined as sinks.

Any data originated from any of the sources is flagged or so-called *tainted*. The environment keeps track of all tainted variables during the propagation through the system. In general propagation *through the system* means transferring and transforming values with functions or assignments, whereby functions use one or more variables as operands. When the value of x is assigned to y ($y = x$) the information is said to *flow* from x to y and is termed *explicit flow*. If the variable x is tainted, the variable y would be tainted after the assignment too. In case of a tainted variable encountering a sink the system would raise an alert.

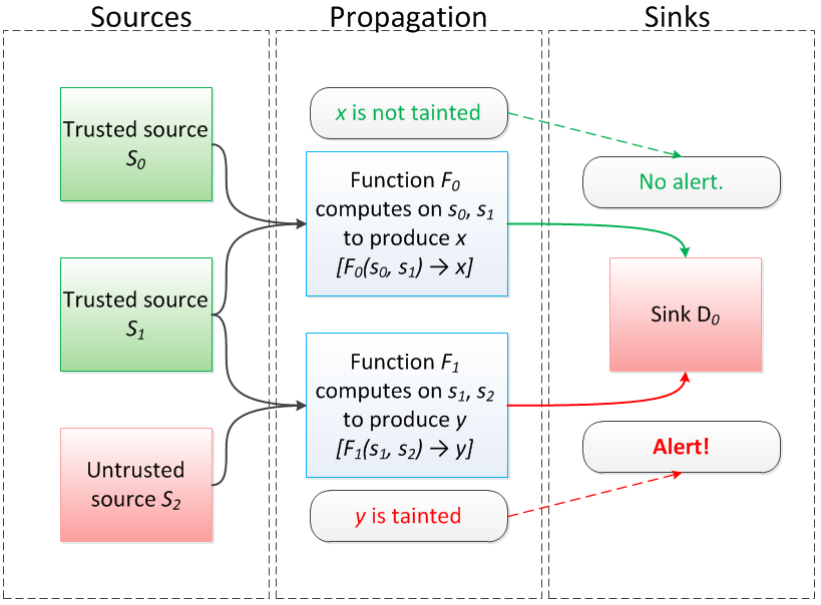


FIGURE 2.3: An illustration of a dynamic taint tracking system.

The programming language family Perl designed by Larry Wall in 1987 [95, p. 1073] introduced a taint tracking system. Any data originating from outside a program itself is marked as tainted. During the execution flow Perl prevents the usage of tainted data as arguments for sensitive functions or instructions [95, pp. 558]. Later on, other interpreted languages as Ruby [88] or Python [45] applied similar approaches.

Static taint tracking has been used to identify buffer overflow, string format, and Structured Query Language (SQL) injection vulnerabilities [30]. However, it comes with major disadvantages. The program's source code is required for static analysis, but malware employs obfuscation or encryption techniques to prevent such an analysis [13]. Additional shortcomings are a high false positive rate [13] and some more general limitations due to undecidability problems [30].

As a consequence, dynamic taint tracking systems have been introduced. Haldar et al. [32] proposed a dynamic taint tracking system for Java which tracks user inputs at runtime to prevent malicious code from being executed, imposing negligible performance

overhead. It focuses on most prevalent attacks on web applications like command injection, cookie poisoning, or cross-site scripting. At the same time, Chen et al. [14] introduced an architectural technique to defeat attacks, based on the notion of pointer's taintedness. Pointers are tainted if user input can be used as the pointer value. An alert is raised whenever a tainted pointer is dereferenced during program execution. Hence, it is effective against memory corruption attacks like buffer overflow, heap corruption, or format string. Panorama, developed by Yin et al. [100], is the first system which tracks data while it propagates through the entire operating system. In fact, this makes it possible to detect and categorize unknown malware. However, it comes with significant performance issues. A variety of approaches have been published in order to improve malware detection and classification [28, 55, 78]. Moreover, it was also possible to leverage dynamic taint tracking that one is able to detect leakage of confidential information. In contrast, Enck et al. presented *TaintDroid* [20], a system-wide dynamic taint tracking and analysis system capable of simultaneously tracking multiple sources of sensitive data. It is implemented within the user space. Section 2.3.4 explains the system's properties in detail.
Another notably property of the proposed dynamic systems is that the taint tracking concept has been transformed from a tool employed to protect from untrusted *inputs* into the opposite: A tool to guard from unintended leakage of sensitive information (*output*).
On the one hand, these approaches helped to overcome the previously mentioned shortcomings of static systems. On the other hand, they are facing new challenges since malware began to embed evasion techniques to determine whether it is under analysis or not [13]. These techniques exploit specific system properties of taint tracking environments. All three logic units (cf. Figure 2.3) within a taint tracking environment have specific weaknesses. Those will be discussed briefly below.

Information is only tainted if it was at some point obtained from a predefined source. Thus, attackers do not necessarily read data from a source. Instead, they could read from indirect sources like trusted applications or not tainted files [13]. Consider a pointer p which points to an untainted file x. If a malware could change p on its behalf it can access a tainted file undetected. Such vulnerable properties may occur inevitably within large systems, notwithstanding the practical threat depends strongly on the system's context [13]. Moreover, large systems run generally the risk of not identifying all sensitive sources.

The major weakness of dynamic taint tracking during propagation is the fact that it

only tracks *explicit flows*. Explicit flows are, as previously exemplified, explicit transfers of values from x to y, such as $x = y$. If y was tainted, the taintedness would haven been transfered to x accordingly. In contrast, implicit flows do not assign variables directly as depicted in Listings 2.3 and 2.4.

```
if(x = 1)
        y := 1;
else
        y := 0;
```

Listing 2.3: Implicit flow example.

Regardless, the value of x has been indirectly transfered to y (Listing 2.3). If x was tainted the variable y would spuriously *not* be tainted.

```
boolean b := false
boolean c := false

        if(! a)
                c := true;

        if(! c)
                b := true;
```

Listing 2.4: Another implicit flow example [54].

An even more complex implicit flow is shown in Listing 2.4. When a is *true* the first statement is *false* and the branch is not executed. As result b correlates with the value of a, even in absence of a direct assignment. The fact of omitting the first branch contains information which is exploited in the second *if* statement to prohibit b from being tainted. Since a affects c and c affects b, it is a transitive enclosure: a is said to affect b.

These false negatives are a known aftermath of the *under-tainting* problem. It is beyond the means of the formerly referenced taint tracking approaches and further addressed in various academic publications [13, 20, 30, 72].
An attacker can propagate an arbitrarily large amount of tainted information without using explicit flows. Nair et al. propose with *Trishul* [54] an approach to overcome this limitation by combining dynamic taint tracking with static concepts. Trishul captures the set of variables involved in a conditional code branch and calculates a list of modified variables for each block. The objects which are modified in any of the possible paths are

tainted regardless whether the path is taken at runtime or not [54]. Graa et al. extend this approach to overcome this weakness for the TaintDroid platform [30] (cf. Section 2.3.4). However, the practical implementation is not yet publicly available.

Another manner of circumventing taint tracking are *side-channel attacks*. They originate from the field of cryptography and have been introduced by Kocher in *Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and other Systems* [40]. Since any entity within a system interacts with and is influenced by its environment, information may be leaked through this indirection, even if it is not intended. This kind of information is called side-channel information, and the attacks on side-channel information are called side-channel attacks. For illustration, Kocher measured the amount of time to perform private key operations to make a qualified guess on the Diffie-Hellman exponents or RSA factors [40]. The Windows malware *W32/MyDoom* uses a code execution timing attack to determine if it is executed in a sandbox (cf. Section 2.3.2) or not [85]. An instance of a side-channel attack, targeted at a dynamic taint tracking system for Android, is given in Section 2.3.4.
In regards of taint tracking the difficulties emerge, that side-channels are easily overlooked during definition of taint sources. Moreover, the known countermeasure of heavy tainting would lead to *over-tainting*. Due to this, side-channel attacks seem to be the hardest weakness to challenge. Publications on the limitations of dynamic taint tracking or information flow control tend to keep it out of scope [13, 30, 54].

In addition to the previously discussed components, the process of identifying and tagging sinks reveals design weaknesses too. As explained beforehand, attackers do not necessarily read data from a tagged source, the same holds true for writing data to sinks. It is infeasible to identify all potential sinks within a complex system in practice. Thus, we are showing in Chapter 4 a successful sensitive information leakage attack on TaintDroid.

As stated in Section 1.3 the improvement of dynamic taint tracking systems by leveraging static facilities to overcome the outlined shortcomings is beyond the scope of this Thesis. Nevertheless, it is crucial to understand the limitations of the solution implemented in TaintDroid respectively DroidBox. Solely in this way, one is able to understand evasion techniques of malware samples under analysis.

2.3.4 TaintDroid

In 2010 Enck et al. presented *TaintDroid*, a system-wide real time dynamic taint tracking system to detect privacy leaks on Android [20]. It extends the Dalvik virtual machine to track the flow of information from a tainted source through third-party applications. TaintDroid's design was inspired by the previously introduced publications [28, 32, 33, 100]. Thus, different challenges had to be addressed due to the limited resources of mobile platforms and the requirement of real-time monitoring.

The Dalvik VM is instrumented to integrate different granularities of taint propagation as described in *TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones* [20] and visualized in Figure 2.4.

- *Variable-level tracking* is realized in the VM interpreter and taints variables in application code.
- TaintDroid uses *message-level tracking* to reduce performance and storage overhead since it tracks IPC messages, not single variables. As stated in Section 2.1 all IPC occurs through binder. Hence, it is implemented directly within the binder component.
- *Method-level tracking* is implemented to taint data, even when it is distributed through native methods. Potential return values are tagged after at least one parameter was tainted. The native library loader is modified so that it exclusively loads trusted system libraries and prohibits application's private third-party libraries.
- The *file-level tracking* assures that information is correctly tainted even after a persistence cycle. It includes database files.

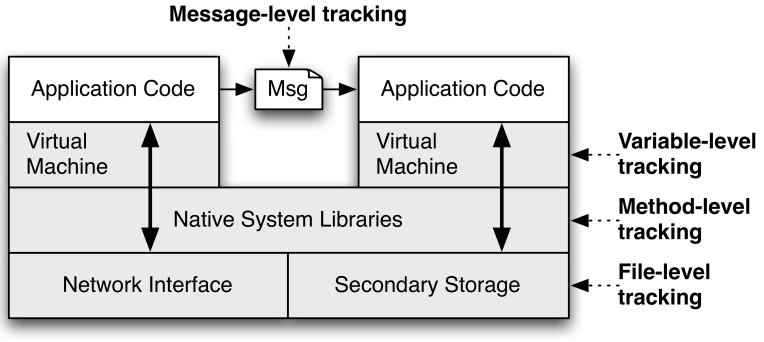


FIGURE 2.4: Multi-level approach of TaintDroid [20].

Figure 2.5 illustrates the TaintDroid architecture within Android. After information is retrieved from a taint source (1) a native method is invoked through the taint interface within the Dalvik VM interpreter. It stores the taint in the virtual taint map (2). The taint tags are propagated according to the data flow (3). When tainted information is used in an IPC transaction, the modified binder ensures (4) the correctness of taint markings in the data transfer object (parcel). The parcel is transfered to the remote binder via the system kernel (5). While it is unpacked, the remote binder assigns the taint tag to all information read from it (6). The remote VM handles the taint marks just as within the trusted application (7). In case tainted information is about to be released through a sink (8,9) the event is reported and an alert is raised.

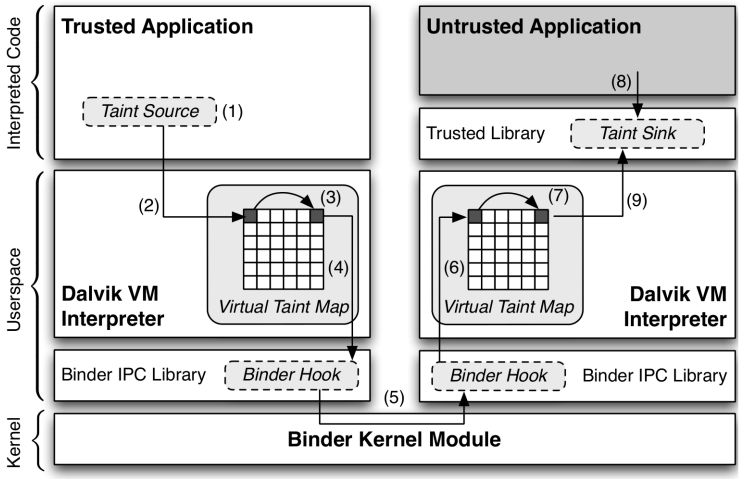


FIGURE 2.5: The architecture of TaintDroid [20].

The authors of TaintDroid categorized all taint sources they have identified as follows (cf. Table 2.3):

- *Low-bandwidth sensors* provide information which change frequently and are used by several applications at the same time. Hence, the OS provides multiplexed access through a manager interface.
- On the contrary, the *high-bandwidth sensors* supply single applications with a large amount of data. Data is shared through data buffers or files, thus TaintDroid places hooks for both, to track microphone and camera data.
- Larger amounts of data are stored in *databases* and made accessible to apps by manager interfaces. By adding tags to database files all data read from will be tainted by design. SMS messages and contacts are tagged in this manner.

Taint Source	**Sensor Type**	**Implementation**
Location sensor	Low-bandwidth	LocationManager
Accelerometer	Low-bandwidth	SensorManager
Microphone	High-bandwidth	Hooked API
Camera	High-bandwidth	Hooked API
Address book	Database	Database file
Phone number	Device identifiers	Hooked API
IMEI	Device identifiers	Hooked API
IMSI	Device identifiers	Hooked API

TABLE 2.3: Overview of TaintDroid sources.

- There is a variety of reasons modern communication devices need to be identified uniquely. Since the information is privacy sensitive it needs to be encountered as taint sources, too. It is accessed through a well defined API on Android, which is instrumented as taint source.

Considering the fact that TaintDroid tracks information within the VM, the sink has to be placed within interpreted code. It was placed directly in the Java framework libraries, where the native socket library is called to provide a network connection. Within TaintDroid, it is the only defined sink.

TaintDroid shares the limitations illustrated in Section 2.3.3. In the following we provide an example to any explained weakness vector.
As already explained above, taint tracking systems can be circumvented by utilizing implicit flows. In Listing 2.5 we demonstrate how to leak information (International Mobile Equipment Identity (IMEI)[2]) obtained from a tainted source (*getDeviceId()*) [72]. Since the single digits of $IMEI$ are not explicit assigned to *result*, it is *not* tainted and can be sent via a socket to a server of choice.

```
String IMEI = getDeviceId();
String result;
for( int i = 0; i < IMEI.length(); i++){
        switch (Integer.valueOf(IMEI.substring(i, i+1))) {
        case 0:
                result = result + "0";
                break;
        [...]
        case 9:
                result = result + "9";
                break;
        }
```

[2] The **IMEI** is an unique identifier for each mobile phone world wide.

```
}
sendOverSocket(result);
```

Listing 2.5: Exploitation of implicit flow in TaintDroid.

An instance of a side-channel attack is presented in Listing 2.6. It leverages the circumstance that reading the system clock is not classified as a sensitive operation. Through this channel information can be transfered in an indirect way. For each digit of the sensitive information (IMEI) the *int* value is taken and the current thread is paused for this amount of time. A time stamp is saved prior and after the process is paused. By subtracting the $startTime$ from the $endTime$ we get $result[i]$ where the value of $IMEI[i]$ equals $result[i]$.

```
String IMEI = getDeviceId();
String result;
for( int i = 0; i < IMEI.length(); i++){
        long startTime = getTime();
        this.sleep(Integer.valueOf(IMEI.substring(i, i+1))*10);
        long endTime = getTime();
        result = result + ((int) endTime - startTime);
}
sendOverSocket(result);
```

Listing 2.6: Side-channel attack on TaintDroid.

In additional we found a communication channel which has not been defined as sink by TaintDroid. It confirms the conclusion in Section 2.3.3 that identifying all taint sources and sinks is an error-prone task. If any tainted data is send as Uniform Resource Locator (URL) parameter through an intent triggered by an adversary, the information leakage is not detected (cf. Listing 2.7).

```
String IMEI = getDeviceId();
Intent browserIntent = new Intent(Intent.ACTION_VIEW,
        Uri.parse("http://www.do.not.hit.sink.de/" + "imei" + id));
startActivity(browserIntent);
```

Listing 2.7: Information leakage through a browser intent.

Sarwar et al. [72], presented practical attacks on TaintDroid. However, the TaintDroid authors clarify in their primarily publication [20] that first of all TaintDroid can be circumvented through leaks via implicit flows indeed. Second, the system does not detect side-channel attacks in any manner. Truly malicious applications can bypass

the taint tracking system and exfiltrate information. Thus, Sarwar et al. keep the question of effective countermeasures unanswered. In contrast, Graa et al. came up with a promising approach, unfortunately it has not been implemented to date. To track implicit flows at runtime they want to add a static analysis module to the Dalvik VM bytecode verifier, which checks instructions at loadtime and considers the gained insight during execution [30].

2.3.5 DroidBox

DroidBox is an Android sandbox developed by Lantz from Lund University in the context of *Google Summer of Code* 2011 [16] in collaboration with *Honeynet* [35] and for his master's Thesis [42]. Because of the constraint that TaintDroid does not come with a set of scripts or other framework features to perform an automated analysis of potential malware samples, DroidBox complements TaintDroid with its analysis framework. Hence, its operation principle and the overall concept of taint tracking is characterized in the previous section. Furthermore it modifies the OS core libraries with the aim to log relevant events during the dynamic analysis. Such events are:

- File read and write operations.
- Dump content of file operations.
- Incoming and outgoing network data.
- Cryptography API operations.
- Sensitive information leaked through the following sinks: Network, file, and SMS.
- Sent SMS and phone calls.
- Started services and loaded classes through DexClassLoader.
- Circumvented permissions.
- Hashes for the analyzed package.

In order to support a human analyst in understanding and assessing a malware sample, DroidBox comes with two behavior visualization techniques proposed in *Visual Analysis of Malware Behavior Using Treemaps and Thread Graphs* [90]. A *treemap* (cf. Figure 2.6) represents a structured tree graph as a nested rectangle map. In analogy to a

tree a branch is represented by a section. Sections are divided into smaller rectangles to denote subbranches or nodes. The API calls performed by the applications are associated with sections as network communication, file interaction, or cryptographic operations. The width of any rectangle is proportional to the percentage of a section's operation occurrence. The higher a section rises, the higher is the operations frequency.

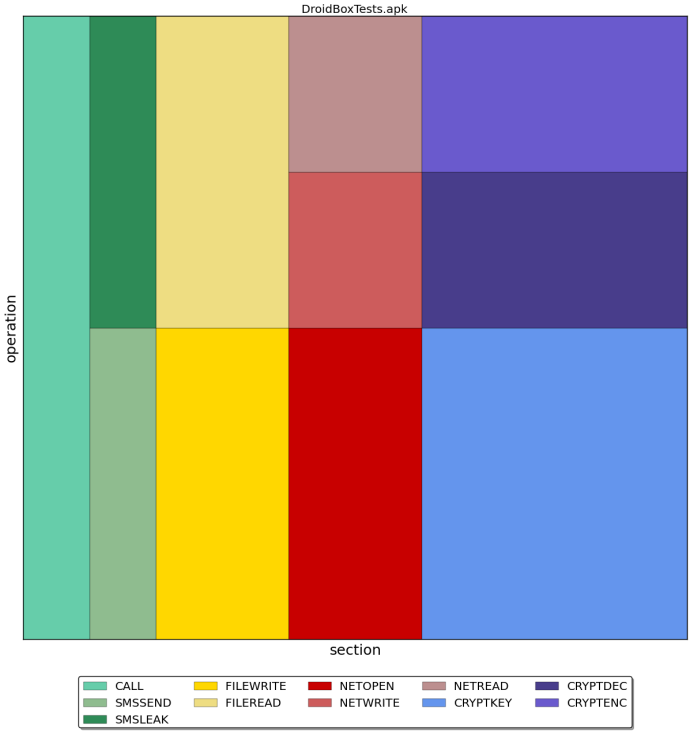


FIGURE 2.6: Sample treemap illustration.

As Figure 2.7 shows, *behavior graphs* visualize the sequence of performed actions (y-axis) distributed over time (x-axis). Since one sample can easily perform thousands of operations during its analysis, a threshold is required to be reached for a single operation in order to include it into the graph.

The DroidBox system is, besides AASandbox [9], the first publicly available open source dynamic analysis framework for Android. On that account it has been used as base for other platforms including TraceDroid [92], SandDroid [10], Mobile-Sandbox [79, 81], and Andrubis [93].

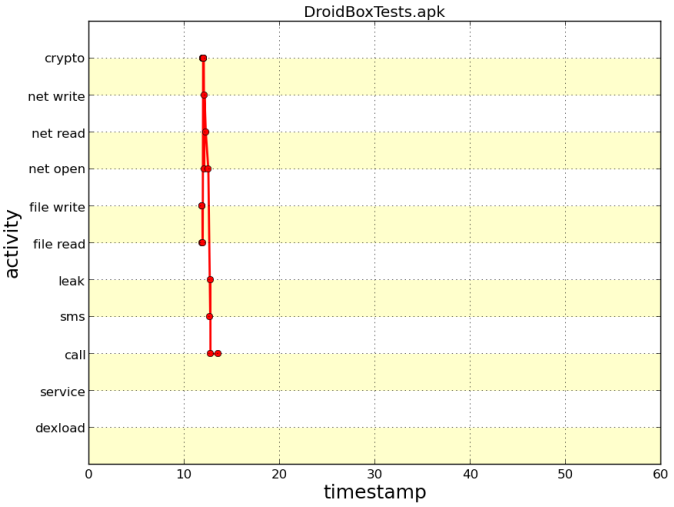


FIGURE 2.7: Sample behavior visualization.

2.4 Summary

This chapter discussed the foundations relevant for the remainder of this Thesis. Initial, the architecture of the Android OS is recapped. Thereafter insights of mobile malware for the Android ecosystem are discussed. In order to analyze such malicious software the proceeding of static and dynamic analysis is given. Finally, internals of a concrete Android sandbox implementation DroidBox is outlined. Moreover, assets and shortcomings are explained in detail. We ensured that the reader is equipped with all required context information to follow the remainder of this Thesis.

Chapter 3

Porting DroidBox

After the introduction sketched the Thesis's background and the previous chapter outlined the foundations, the reader should be familiar with the concept of dynamic malware analysis for Android and understand the necessity to advance it.
This part of the Thesis focuses on porting DroidBox to a more recent version of Android. Section 3.1 gives an overview of the initial situation of the existing Android sandbox environments and its versions. The subsequent paragraph covers details on the proof of correctness for DroidBox 4.1. In Section 3.3 the actual porting of DroidBox to Android version 4.4 is reasoned. Finally, we summarize the still present shortcomings and discuss the lessons learned.

3.1 Related Work and Initial Situation

As already concluded in the introduction, there is a strong demand for a dynamic analysis environment which runs up-to-date applications. At the time the current version of Android is 4.4 alias *KitKat*, released in October 2013. When the work for this Thesis began, DroidBox was available in version 2.3 and TaintDroid in version 4.1. The version numbers of both environments correspond to the Android version they are based on. An overview of relevant Android versions along with more precise API version codes can be found in Table 2.2.
On the one hand, DroidBox 2.3 can be considered as comprehensive and one of the only publicly available environments, but on the other hand, it is fairly outdated and not capable of analyzing applications deployed for recent versions of the Android OS. Since it bases on TaintDroid which is existent in version 4.1, the porting comes in as an

obvious option to overcome the sketched limitations. Unfortunately, this issue was not only asserted by us. In October 2013, the source code for DroidBox 4.1 was published by hispasec[1] employees. We decided to take advantage of the synergy effect by using it as a basis for further modifications. Therefore, proving the correctness of their work (cf. Section 3.2) is mandatory. A supplementary goal is to contribute with an answer to the question whether it is feasible to automate and generalize the porting process or not. On that account, DroidBox 4.1 is ported to 4.4 additionally, in case the correctness can be evaluated successfully. As far as we are concerned it is even the more challenging part, because all transformations primarily conducted by TaintDroid are also subject of such a port. However, Section 3.3 elaborates the porting by exemplifying meaningful code samples.
As outlined in Section 2.3.5, the dynamic analysis sandbox DroidBox leverages the Android emulator. Hence, the original Android source code needs to be obtained, modified, and built in order to run either TaintDroid or DroidBox within the emulator.

Porting security extensions of an OS are rather a practical task than of interest for the research community. Thus, a limited body of research is accessible. However, the feasibility of automation is a similar problem as the one Android smartphone vendors are facing. With any new release their customized OS extensions are subject to a comparable question: How to adapt changes to the new Android version as efficient as possible in order to provide an update for customers. A recent review of the situation suggests that no satisfactory solution was found yet [43]. Another Master's Thesis, examining the porting of a DroidBox version leaves it unanswered [12].
Enck et al. presented their work on TaintDroid in "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones." [20]. Its essence is previously covered by the Subsections 2.3.3 and 2.3.4.

3.2 Correctness of DroidBox 4.1

Android 4.1 alias *Jelly Bean* has been released in June 2012. Just six month later a new version of TaintDroid was announced. It finally got integrated into DroidBox in October 2013. As aforementioned, the port was performed by employees of hispasec and the source code patch files are provided online[2].
In order to run DroidBox 4.1, a custom build is required. First, the original Android

[1] **hispasec** is a Spanish IT security and security information provider (`www.hispasec.com/`).
[2] **DroidBox download** at Google Code (`code.google.com/p/droidbox/downloads/list`).

code needs to be obtained. Google maintains a tool called *repo*[3] which is based on the distributed revision control system *git*[4] to manage all Android source code repositories and deliver it to the developers.
Second, the patches must be applied to the downloaded Android code. After setting up the build environment the build process can be initiated. A detailed step-by-step instruction manual is given in Appendix A. Since such a build process is not trivial and some prerequisites are obligated to meet, we uploaded a ready-to-use emulator image file to the publicly available DroidBox Google Code repository.

To ensure an accurate analysis outcome it is crucial to evaluate the new version prior to its integration into an existing sandbox framework. By proving the correctness we mean that malicious actions performed only appear in the analysis result, if and only if the related action was indeed performed by the analyzed app. Furthermore, we expect the implemented extensions not to influence the stability of the OS in any manner.
We assume the correctness of DroidBox 2.3 as well as Android 4.1. DroidBox 2.3 has been deployed in several sandboxes all over the security community so that potential misbehavior would have been identified and eliminated meanwhile. The same holds true for the Android OS, but even on a larger scale since the number of users and developers is significantly higher. Our test methodology consists of three independent approaches with which we seek to assure the correctness of DroidBox 4.1 as follows:

- The Android build environment holds a set of automated basic tests which are executed by default right after the build terminates.
- In the context of this Thesis we developed a Python-based test environment that runs an arbitrary collection of apps in DroidBox 2.3 and 4.1 consecutively. The result would unfold potential anomalies.
- A manual inspection of a meaningful sample set yields the correctness of detected suspicious actions which are monitored by DroidBox.

Automated smoke test[5] are widely deployed in practice to identify critical bugs as soon as the build process finishes. The build tests of Android verify that all crucial system components work as expected. All components which are part of the application framework and the application layer are tested with JUnit[6] test cases. Since all tests are

[3] **Repo**, an extension of git to manage multiple repositories at the same time (`code.google.com/p/git-repo/`).
[4] **Git**, an open source distributed revision control system (`http://git-scm.com/`).
[5] **Smoke test** refers to tests of systems intended to determine readiness for more complex testing.
[6] **JUnit** is a unit testing framework for Java (`junit.org/`).

executed by default after the build operation there is no need for customizing it. DroidBox 4.1 passed all test cases. Ergo, we are confident that it does not contain critical bugs which endanger our automated analysis environment.

In order to compare the analysis results of a unique app, we developed an automated test environment. First, it performs the analysis with DroidBox 2.3 followed by DroidBox 4.1. Each sample residing in a specified directory is installed into the emulator and executed for 60 seconds. The result is stored into a database and written into a human readable log file.
Both environments are identical except for the OS instance running within the emulator. We adapted the sandbox control scripts of 4.1 to operate with the legacy version. The scripts utilize the Android Debug Bridge (ADB)[7] to install and run the app inward the emulator. Besides, ADB serves as a communication channel between the emulator and the host by transmitting log output in real time. Security violations and suspicious activities are logged and stored within an eligible data structure. After expiration of the execution time the emulator is shut down. Any change made to the emulator is discarded by deleting the file containing user-data. For each sample a new instance of the emulator is spawned to guarantee independent results. User input is generated by the monkeyrunner[8] tool.

To verify the correctness of DroidBox 4.1, we chose 20 samples randomly from a set of confirmed malicious applications originating from Andrubis [93]. The results of the two different environments can be easily compared manually because we designed the log file representation exclusively for that purpose. As outcome we found that 18 of the 20 samples (90%) yield an identical result. However, the results of two samples diverged from each other. Thus, we manually inspected both with the bottom line that we could not observe any differences. Hence, we can just speculate about the reasons. Occasionally, unreproducible ADB connection errors or invalid monkeyrunner inputs were observed in the past.
Nevertheless, we state that the tests successfully evidenced the correct working of DroidBox 4.1 for the used test set since the minor deviations are irreproducible.

Finally, we seek to validate the correctness of monitored behavior. A set of malware samples is selected and listed in Table 3.1. Each sample performs a distinct suspicious operation about which we are aware of in advance. Hence, it is contrivable to evaluate if the ported version of DroidBox correctly detects those actions.

[7] The **Android Debug Bridge** is a tool to communicate with an emulator instance.
[8] The **monkeyrunner** tool provides an API for controlling an Android device or emulator.

Name	Checked Feature	SHA Hash
Simhosy	A, B	73589c7a4044bd3ea9403f62c6afd54fe1bd90dc
Fakemart	A, C	b45d969a8fd1d3fb2a787cab8460b54088d89770
Lien	D	f04dff1859c9cf43260020b1e4dbbe979fe1bcc1
FakeAV.D	E, F	616b37d70819d99e53c007ca00d4d599099a9cd6
Wooboo.A	G	d0e2799bacc9a59e91ec80bec81f1d24d856664b
MouaBad.P	H	5ad4a348381bbfeafd71661e2ccda63c2ef960e7

TABLE 3.1: Malware used to evaluate DroidBox 4.1 (**A** = network operations, **B** = started services, **C** = send SMS, **D** = registered broadcast receivers, **E** = file operations, **F** = taint tracking, **G** = cryptography usage, **H** = phone calls).

The contemporary validation collection accommodates six instances of different malware families. They emerged in the wild in 2013 and either innovated new features or they are representative for a certain kind of malware. All pieces are confirmed malicious by VirusTotal's[9] analysis results.

Fakemart (cf. Figure 3.1a), for instance, sends SMS messages to premium numbers on the user's expense. It poses as a third-party market but does not provide any of its advertised services. At first the malware sample sends client information to a web server and as soon as it receives the response premium SMS messages are sent. Likewise, *FakeAV.D* tries to trick the user by pretending to offer an AV service as illustrated in Figure 3.1b. In reality, user information and device identifiers are sent right after the application is started.
The mobile malware **Mouabad.P** is noteworthy because it can initiate a call without user intervention to premium numbers. Thus, it is the first instance with such capabilities and acts stealthy. In order to perform phone calls it waits until the screen turns off and ends them as soon as the user interacts with the smartphone again.

Analogous to the detection features of DroidBox (cf. Section 2.3.5), the verified operations are abridged in the following:

- *File operations* are tested by **FakeAV.D**. The malware comes with its own database which is stored in the apps private data folder. The database file creation operation is recognized by DroidBox 2.3 and 4.1. App specific settings are stored in preference Extensible Markup Language (XML) files in the private data folder. Each modification is realized respectively and assumed to work correctly.

- *Network transaction* monitoring's correctness is demonstrated by the malware samples **Simhosy** and **Fakemart**. Both send device specific information to a web

[9] **VirusTotal** provides access to up to 48 anti virus (AV) solutions simultaneously (`virustotal.com`).

server. In both cases each version of DroidBox correctly recorded the transmitted data. However, the destination Internet Protocol (IP) address differs. The malware uses URLs (e.g. `mathissarox.myartsonline.com`) which are mapped to a range of IP addresses. Finally, we could yet show the accuracy.

- *Cryptography API usage* detection is inspected through the **Wooboo.A** malware sample. It transmits encrypted data over the network. Both analysis results list the identical key. Note that we did not attempt to decrypt the sent data with the extracted key.

- *Send SMS message* detection is evaluated by **Fakemart**. As outlined above, **Fakemart** sends SMS messages to premium rate phone numbers. DroidBox 4.1 successfully detected the same message body (AP) and destination number (81238) as DroidBox 2.3.

- *Outgoing phone calls* are observed with **MouaBad.P**. The called premium numbers are correctly noticed by DroiBox 2.3 and DroidBox 4.1 likewise.

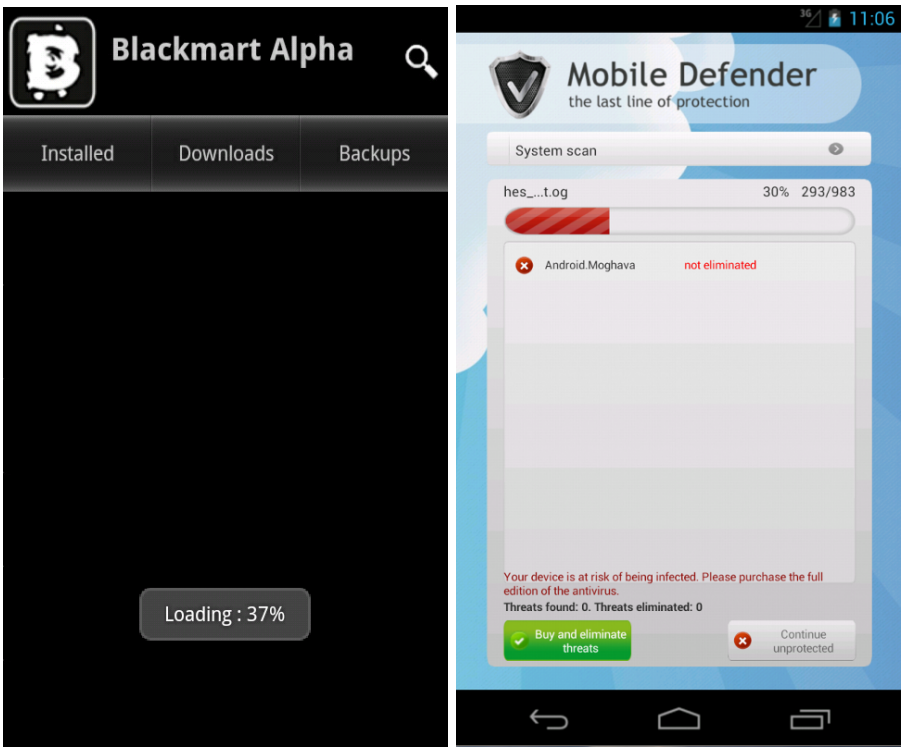
- *Started system services* are verified by **Simhosy** and **Wooboo.A**. The service *.DownLoadService* is started by **Simhosy** and *.CompatibilityServiceX* by **Wooboo.A**. Both were detected on the new DroidBox version and can thereby be considered as correct.

- Broadcast receiver monitoring's verification is performed by instrumenting the **Lien** sample. The correct detection of two registered broadcast receivers states that this part of DroidBox 4.1 works as expected.

Consequently, we could not observe any relevant anomalies in any of the applied test scenarios. The realized deviations could be explained doubtless. Hence, we consider the ported DroidBox 4.1 as successfully evaluated.

3.3 Porting DroidBox to Android 4.4

Despite the fact that DroidBox 4.1 is fully functional, as demonstrated in the previous section, a port to the latest Android version is a valuable contribution for the community. Android 4.4 is a major release and offers new features for application developers as well as end-users. Thus, it is expected to gain a relevant market share over time, as new devices are sold and old ones updated by the manufactures.



(A) Fakemart. (B) FakeAV.D.

FIGURE 3.1: Graphical user interfaces of two malware samples.

In the future, developed apps will target the new platform since the new features can only be accessed by the updated API. The downside caused by this innovation process is the lack of backward compatibility. New apps do not necessarily run on outdated OS versions. Although malware authors aspire to target a preferably high number of devices and commonly do not employ latest features, an up-to-date analysis environment is desirable. DroidBox deployments which are disposable online through websites for end users will be faced cutting-edge apps nevertheless. Users may want to verify if a new application is truly benign.

Additionally ART, a new Android runtime environment has been introduced with the update. It is designed to accelerate app execution performance and system stability. The DalvikVM supports Just In Time (JIT) compiling. Each time an application is launched, and therefore loaded into the DVM, its bytecode is compiled into the targets platform machine code (e.g. ARM or x86). This procedure is inevitable because bytecode is platform independently. In contrast, ART introduces the approach to compile apps at the time of installation once and not at each launch again [68]. This modification is suspected to reduce the start-up time noticeable and improve the preformance in general. The machine code is saved as OAT file for future use. Google announced its plan to replace the DalvikVM in an upcoming release by the ART [61]. Hence, TaintDroid

cannot be used as basis for DroidBox anymore. A new taint tracking system has to be designed from scratch.

The port of DroidBox 4.1 to Android 4.4 is a more complex task than simply porting TaintDroid to DroidBox. All extensions applied to Android by TaintDroid have to be ported in addition. This is mainly the taint tracking system with all its taint sources, propagation tracking, and the taint sinks. In order to estimate the workload, the relevant modifications had to be identified. They are outlined for each version subsequently.

Android 4.2 [58]:

- Global systems settings were stored in `Settings.System` to date. Some are defined as read-only now. Besides, device identifiers are moved to `Settings.Global`. These changes have impact on taint sources for the device identifiers.
- Android introduced multiple user spaces, mostly for tablet devices. Therefore the API to access the private user storage is modified. These changes affect the taint sources for files.

Android 4.3 [59]:

- Two new rotation sensors allow access to raw data instead of preprocessed estimated bias values. If the sensor's data is considered as sensitive information new taint sources must be defined.
- A new contacts provider interface demands adaption of the taint source for the contacts.
- Android 4.3 offers a custom provider in the key store facility for app-private keys. In order to encrypt communication or data of malware an interface to track the usage might be of value.

Android 4.4 [60]:

- The previous SMS provider API allowed apps SMS messages. The updated provider specifies an default SMS app and regulates the access more strict. This modification impacts the SMS tracking capabilities of DroidBox.
- External storage access API was extended by an method to retrieve data from shared storage. Adjustments for the taint tracking system are required to be undertaken.

- A step detector and a step counter sensor interface which supports built-in sensor hardware. In order to keep track of queried information they have to be defined as taint source.

Considering the list of changes we comprehend the Android modifications on a top level layer. In the next step, the porting procedure on the source code file level is presented. The assumption is that OS files which are not extended by DroidBox nor touched by the Android update to version 4.4 can be ignored. On that account all files whether extended by DroidBox or updated by 4.4 need to be identified. The applied changes of DroidBox need to be merged in the corresponding Android 4.4 files. Also, files which are altered by DroidBox and are not element of the Android source tree anymore are subject to further investigation in order to appraise its impact.
In practice, a list of all source code files with their associated Secure Hash Algorithm (SHA) hash is generated for all three source code repositories: Android 4.1, DroidBox 4.1, and Android 4.4. The SHA is a cryptographic hash function which maps data of arbitrary length to a result sum of a fixed length. If only one bit of the input is altered the resulting hash deviates significantly. Hence, the generated file lists with the hashes are utilized to distinguish whether a file is changed between two different source code repositories or not. At first, a change list for Android 4.1 and DroidBox 4.1 is created. It enlists all files which were extended by DroidBox. At second, a list of changes between 4.1 and 4.4 is generated which indicates the updated files. Finally, the intersection set of both lists contains all files which are relevant for DroidBox and were touched during the update process. Moreover, all source code files which are not part of Android 4.4 anymore are included. This proceeding is depicted in Figure 3.2.
In total, the resulting list consists of 116 source code files, more specifically, 40 C/C++, 35 Java, 5 Assembler, and some Hyper Text Markup Language (HTML) files. The changed lines of code within a file vary from one line of code to almost the entire file. In some cases the file does not exist in the newer Android version anymore. For a better understanding of the port, modifications are exemplified in Subsection 3.3.1.

We developed a toolchain to automate the process of creating the list of files which need to be examined manually due to changes of the OS. A shell script (*genhashlists.sh*) first generates all hashes for all files of each repository. These lists are compared by another shell script (*gendiffs.sh*) to identify all deviating files. Therefore, we utilize the Unix program diff[10]. Its output is cleaned by the Python program *clean_diff.py* to filter the relevant entries. As a result, a list containing all paths and files mandatory

[10] **diff** is an Unix program for text file comparison that outputs the differences between two files.

for the porting process is generated. All scripts and programs can be found on the accompanying DVD (cf. Appendix D).

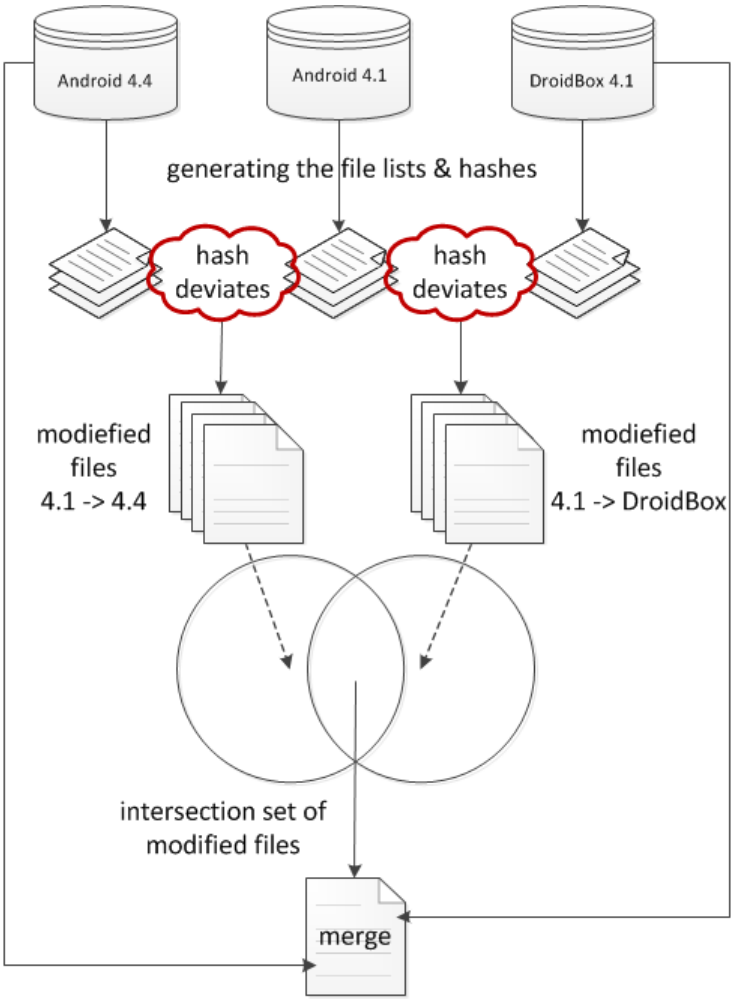


FIGURE 3.2: Illustration of the DroidBox porting procedure.

After the port is completed the adjustments are provided as git patch files. Thence we instrumented the repo tool once again. The resulting patches can be applied together with the DroidBox 4.1 patches on top of an untouched Android 4.4 repository.

Eventually, we drafted the procedure of identifying the modifications required for a port. The general system changes which influence DroidBox on the one hand, as well as the file level modifications on the other hand. After conducting the adjustments we described how patch files are extracted and applied to the clean Android 4.4 repository. The last open issue about how the porting is performed is addressed in the following subsection. Additionally, we discuss the feasibility of automating the entire porting process.

3.3.1 Modifications

Within this paragraph we seek to utter insight into the actually course of porting. Thus, some meaningful practical examples are presented.
Generally, the TaintDroid authors employed a clean coding style for their OS extension. The Android build system supports a build specification file *buildspec.mk* which defines build parameters and properties. It allows to include or exclude the taint tracking system from the compile process depending on the value of the flag `WITH_TAINT_TRACKING`. Within the C/C++ source code files all TaintDroid related modifications are enclosed with the preprocessor directive `#ifdef` to deploy conditional compilation as depicted in Listing 3.1.

```
#ifdef WITH_TAINT_TRACKING
    The modified or extended portion of C/C++ taint tracking code goes here.
#endif
```

LISTING 3.1: A preprocessor directive to enable/disable taint tracking at compile time.

In addition, every single modification in Java is enclosed in comments as exemplified in Listing 3.2. In contrast to the C/C++ preprocessor directives this does not yield in a conditional compilation. It allows to gain a prompt overview of changes within a source code file by a text editor search. Deplorably, DroidBox modifications are not clearly indicated.

```
// begin WITH_TAINT_TRACKING
    The modified or extended portion of Java taint tracking code goes here.
// end WITH_TAINT_TRACKING
```

LISTING 3.2: Enclosed comment for TaintDroid modifications.

The general approach is to work through the list from top to bottom and inspect each file manually. Ergo, each file is compared with a suitable graphical text editor which displays the Android 4.4 and DroidBox 4.1 file version at a glance. The TaintDroid modifications are easily identifiable due to the comment notation. The differences and the necessary steps are straight forward for an expert with programming experience in most cases. Nevertheless, it is a decision on a by-case basis and a universal course of action cannot be given in an useful manner. For that reason, some concrete examples are itemized afterwards.

The library core file *MemoryBlock.java*[11] for managing memory blocks is modified by DroidBox as well as Android 4.4. The data type `MemoryBlock` offers methods to store different variables of common data types within a memory block at a defined offset. The taint tracking system is compelled to keep track of such variables. Android 4.4 is adapted to support 64 bit memory addresses. On that account, the word width of the address variable is extended from *integer* to *long*. The modifications are in the same file and method but not in the same line.

The Portable Operating System Interface (POSIX) standard has been introduced by the Institute of Electrical and Electronics Engineers (IEEE) for maintaining interoperability between OSs. Android provides a POSIX API with *Posix.java*[12]. Between Android 4.1 and 4.4 it was extended. Due to its data exchange methods a lot of taint tracking related functionality is accommodated. Android 4.4 introduces a new exception: The `SocketException` is a reason why several method signatures are altered and therefore DroidBox and Android changes affect the same lines of code.

In accordance with the postulated changes in the SMS messaging system the default SMS managing instance *SmsManager.java*[13] was removed. For continuous tracking of incoming and outgoing SMS the modified API is extended.

There are also source code files which merely contain very few lines of DroidBox specific code. If such a file is altered by an OS update the porting effort is marginal. The DVM C/C++ header file for interpreter definitions *InterpDefs.h*[14] was extended with 4.4 slightly. Since the DroidBox code is in another method and line the required adaption are rather obvious.

Another kind of alteration is the rearrangement of source code subtrees. The *binder.c* was moved from *"./frameworks/base/cmds/servicemanager/"* to *"./frameworks/native /cmds/servicemanager"*. Source code related modifications are not needed.

As already outlined there is a variety of factors for OS source code modifications. At first, new features for developers and end-users breed new software components. Depending on the relevance for malicious appliance the developed code demands extension for the sandbox. This is achieved by reviewing the Android change logs.

Moreover, frequent changes to improve features, the internal structure of the OS, or fix bugs are constantly undertaken. Hence, it comes to unpredictable modifications which have to been taken in consideration for a port of DroidBox. It is crucial to identify all OS files which are adapted by the Android update and DroidBox likewise. The generated

[11] Android repository path: ./libcore/luni/src/main/java/java/nio/MemoryBlock.java
[12] Android repository path: ./libcore/luni/src/main/java/libcore/io/Posix.java
[13] Android repository path: ./frameworks/base/telephony/java/android/telephony/SmsManager.java
[14] Android repository path: ./dalvik/vm/interp/InterpDefs.h

list is processed file by file and the differences are compared manually. As demonstrated above, the adjustments regarding the source code are a decision on a by-case basis that requires a developer aware of the environment. Furthermore, in rare cases OS components are removed and a replacement is added within an other component, which necessitates context awareness as well.
Terminally, the question of automating the port process remains. As we have shown, it is practicable to generate a list of files which need to be examined automatically. This inevitable procedure facilitates the work already significant. Certainly, the other essential part requires at least some manual work depending on the category of modification. Assuming the DroidBox related code is clearly labeled and the affected lines are not subject to the Android update it might be practicable to merge both code files automatically. Nevertheless, it is still expedient to verify the result by a human expert. An expanded API might require further changes concerning the sandbox which are too complex to be practically implemented in an reasonable way. Structural environment adjustments are hard to detect and transform automated. An adequate example is the 64-bit architecture extension. It is not challenging for a human programmer to recognize the change of a data type even if it is within the scope of a DroidBox enhancement. Thus, this cannot be mapped by software and is beyond practical utility.

Ultimately, porting DroidBox to the latest Android version can only be semi-automated because of introduced or modified components which require context awareness. This complexity cannot be represented with justifiable effort in a practically applicable model.

3.3.2 Shortcomings

Despite the general limitations of dynamic taint analysis detailed in Section 2.3.3, further drawbacks are identified within the current implementation.

As mentioned in Section 2.3.4, browser intents can be instrumented to bypass the socket taint sink and leak information to an arbitrary server. This holds true for the port of DroidBox to Android 4.4 comparably.
On grounds of the SMS message handling component modifications the DroidBox 4.4 lacks a fully functional implementation. The standard SMS application is able to handle messages indeed, though the new design which allows several event receivers is not supported yet.
Another internal adaption affects the DVM string processing. Since the string data type is widely used within the OS itself and other applications, errors have a particularly

negative impact. Without the application of these changes the unrestrained operation is questionable yet.
As initially depicted, Android 4.4 introduced a new runtime environment. The taint tracking system is not compatible with the ART. For the moment this fact is neglectable since the OS decides which runtime is used. As soon as ART is the default environment this will be a considerably drawback for upcoming releases.
After all, a meaningful test to proof the correctness of the ported DroidBox following the procedure of Section 3.2 is still pending. Hence, DroidBox 4.4 cannot be considered fully functional yet.

3.4 Summary

This chapter intensively dealt with aspects of porting DroidBox to a more recent version of Android. As we could demonstrate there is a strong demand for a dynamic analysis environment in order to examine up-to-date Android applications and malware.

We tuned in to two independent approaches. The already available version of DroidBox 4.1 was proved to be correct by three independent test scenarios. First, the Android build tests as smoke test. At second, we verified the correctness with an automated analysis framework. Finally, malware with known malicious behavior was utilized to investigate defined functionality of DroidBox. All tests results are positive, thus we consider the DroidBox 4.1 as fully functional.

Furthermore, we sought to port the successfully verified version of DroidBox 4.1 to the latest version of Android 4.4. This can be considered as a even more complex task than porting TaintDroid 4.1 to DroidBox 4.1 which originally was planned (cf. Section 1.2). The thereby developed toolchain semi-automates the process of identifying the relevant files for a further manual investigation. However, we conclude that a fully automated environment for porting DroidBox to further Android versions cannot be applied with reasonable effort. Anyhow, insight in the manual process is given. A final review to assure the correctness is still pending. The created git patch files are provided at the accompanying DVD.

Consecutively, we utilized DroidBox 4.1 as basis for further modifications. In the next chapter an overview of emulator detection strategies and implemented countermeasures is given.

Chapter 4

Android Sandbox Detection

As outlined in Chapter 1, the Internet has turned into an economic factor which also appeals to miscreants. Thus, the total number of malware increases notably, for PCs and since a decade for mobile devices likewise.

Chapter 2 clarifies the need for dynamic analysis environments, imposes the sandbox concept, and explains the foundations of DroidBox. When unknown malware appears in the wild, researches as well as security vendors, attempt to analyze its properties. At the one hand, in order to devise detection capabilities and at the other hand, to estimate its impact.

Static code analysis is one facility to examine malware without actually running it. The major challenge is a growing number of malware that employs obfuscation techniques. Thereby, they aim to disguise the malicious portions of source code from the analysts. Dynamic analysis is able to overcome code obfuscation by executing it within an isolated environment. Sandboxes originally targeted x86 applications [5, 99], but with the rise of smartphones mobile sandboxes have been introduced.

The primal concept of sandboxing is to execute untrusted or malignant code within a controlled, safe, and realistic environment. Sandboxes generally utilize virtualized and emulated environments such as VMware [94], VirtualBox [64], or QEMU [7]. As defined by Popek and Goldberg [67], "*A virtual environment is taken to be an efficient, isolated duplicate of the real machine*". Commonly, it is distinguished between two types of virtual environments. *Virtual Machine Monitor* (VMM) refers to an environment which is identical to the host machine's architecture for any executed application. A significant number of instructions is executed on the real underlaying hardware without interception of the VMM. The only exceptions are differences caused by timing dependencies and the

availability of system resources. However, the VMM is in total control of all provided resources at any time. In contrast, *emulators* simulate the hardware behavior completely in software and do not run code directly on the host. Machine instructions are handled by the emulated environment and translated into the corresponding instruction set suited for the real hardware. This allows an emulation of systems with an architecture different than the host's. A notable drawback is the performance penalty due to the fact that each instruction has to be interpreted at runtime. Nevertheless, the general belief is that system emulators are superior to VMs for the purpose of dynamic malware analysis [69]. A general advantage of virtual environments is that they are tightly controlled. The analyst can pause, revert, or restart the system at any point of time. It prevents from time-consuming reinstalls for each investigation. Yet, another benefit results from the isolation: Malware can only affect the virtual instead of the real underlying system. The Android emulator is leveraged as sandbox for dynamic malware analysis. It uses the open source QEMU solution which supports full system emulation for the ARM architecture. For web-based analysis environments it is embedded into a framework which enables an automated injection of suspicious apps into the system via a web front-end. Several of such web-based analysis services have been deployed within the last years. In particular, TraceDroid [92], SandDroid [10], Mobile-Sandbox [79, 81], and Andrubis [93]. As basis for all these environments serves DroidBox (cf. Section 2.3.5) with its underlaying QEMU.

The sample under analysis is supposed to behave exactly as it would onto a real device. Therefore, it is desirable to grant no opportunity for malicious code to detect the presence of a virtual environment. Unfortunately, malware can indeed determine that it is running within a sandbox. In order to thwart the examination and cover its vicious intention from the analyst, it simply adapts the own behavior to be considered as benign. Even though this behavior is not commonly observable at the moment, we suspect a profound mutation in the future as seen on the Windows platform [18].
However, before going into details, related work in the field of sandbox detection is elaborated in Section 4.1. In the following paragraph (cf. Section 4.2), we identify a wide range of methods to reveal sandboxes for Android and organize them according to our developed taxonomy. Therefore, we consider different system layers and leverage mobile environment specific characteristics.
Thereafter, we present countermeasures to mitigate evasion on Android in Section 4.3.

To this end, we supplement DroidBox with extensions to become undetectable by malicious code. The preceding chapter introduced and verified DroidBox 4.1 as fully functional and serves as basis for our implementations.
Finally, we recapitulate the found sandbox detection methods and the implemented preventive OS adjustments in Section 4.4.

4.1 Related Work

This monograph reviews the current state of research in the field of sandbox detection. Foremost, publications on general VM and emulator detection are briefly discussed, followed by Android sandbox related work.

Upon the rise of VM technologies in various areas of computer science it was also considered for secure computing soon. Robin and Irvine [70], evaluate the security of virtualization techniques and identify methods to detect the presence of a virtual environment. After VMs became common for dynamic malware analysis Ferrie [26], a Symantec Researcher, observes the permanent growth of malware samples that are intentionally sensitive to their presence. Consequently, he presents a multiplicity of detection techniques for all major x86 products, including QEMU. The bottom line of his conclusion is that software virtual machine emulators can be implemented, at least in theory, to reach the point where VM detection is unreliable.
Raffetseder et al. [69] seek to verify if the general assumption that system emulators are more difficult to detect than VMs is justified. They state, as Ferrie, that it is in fact feasible to adapt system emulators to mitigate specific detection methods.
The first taxonomy of evasion techniques for the x86 environment is presented by Chen et al. [15]. Detection methods are clustered by layers of the computer systems with varying levels of difficulty to be performed and the reliability of the outcome. However, the question for improvements to mitigate detection remains unanswered.
One of the first publications addressing detection approaches for Android simply retrieves hardware identification strings which deviate from those of real devices [84].
Along the release of the DroidBox system, the accompanying publication sketches the issue and unveils that the emulator's phone number, among other identifiers, remains equal for each deployment [42].
More sophisticated and general attempts for sandbox detection rely on QEMU's architecture in general. Matenaar and Schulz [48], developed timing measurement of scheduling behavior leveraging the binary translation between real devices and QEMU. In almost

the same manner Schulz [73] observers low-level caching behavior to determine the execution environment to be virtual or not.
In 2012, Google released Bouncer in order to protect the Play Store from malware by performing automated dynamic analysis before distributing new apps [44]. Short after the release, Oberheide and Miller [56], as well as Percoco and Schulte [65], demonstrated how to circumvent sandbox analysis techniques of Google's Play Store.

In the following two sections, novel Android sandbox detection strategies as well as detection countermeasures are introduced. Moreover, already undertaken research will be considered for developing a comprehensive taxonomy.

4.2 Sandbox Detection Approaches

Within the consecutive section we provide an overview of different mechanisms for Android applications to determine the presence of a sandbox. A variety of approaches have been published and are recapitulated in the previous paragraph. However, all existing detection approaches have one in common: The assumption that there are always properties which distinguish sandboxes from real devices.
While a great deal of attention has been given to detect virtualized or emulated environments, almost no research focuses on general system environment related characteristics or user-data. In accordance with Locard's principle, Kirk [39] expressed that "*Physical evidence cannot be wrong, it cannot perjure itself, it cannot be wholly absent. Only human failure to find it, study and understand it, can diminish its value.*". Consequently, we argue that in any analysis system component meaningful evidence can be found, especially in the context of mobile devices.
Smartphones, as opposed to ordinary computer systems, come with unique features, which permit a more detailed context awareness. Due to the peculiarities of small portable computational devices they are continuously in motion. This is constantly encompassed by a multiplicity of sensory microchips such as acceleration, device orientation, or step counter sensors. Additionally, devices are aware of the current absolute global position. Smartphones are connected to at least one communication interface most of the time. Generally speaking, the device is aware of the current state of each communication interface and receives updates on changes frequently. Many context information accrue as a result of the numerous user interactions. Not only state changes such as unlocking the screen, touch and scroll events, or modified audibility, also generated user-data in applications or arbitrary preferences.

We stress that the absence or deviation of the described features which allow context awareness can be exploited to detect Android sandboxes. As concrete sandbox implementation we utilize the DroidBox version verified in Section 3.2. Our approach compares a range of DroidBox properties to real devices with the aim to identify differences. In order to obtain meaningful results two Android phones are chosen: A *Nexus S* and a *Nexus 5* which both run *Vanilla Android*[1]. Hence, the Android version of DroidBox and the devices are identical, except of the sandbox extensions. Furthermore, both devices have been effectively in use on a daily basis to assure realistic reference data. The detection vectors found will be discussed in the next subsections successively.

Since none of the related publications groups or generalizes their findings so that it would match our complete system contemplation, we present a taxonomy suitable for mobile OSs. In our taxonomy, we classify the techniques by the system abstraction at which they target to detect the sandbox as follows: *Emulator Related Detection*, *Environment Related Detection*, and *User Input Related Detection*. A more detailed definition for each group is provided within the corresponding paragraph.

4.2.1 Emulator Related Detection

According to the definition of an emulator, any piece of code, whether executed in the emulator or on a real device, should lead to the same results. Yet, even the intermediate results are expected to be identical. If any difference is observable the presence of a sandbox is assumed. Hence, this group is called *Emulator Related Detection*. Initially we list some general examples [15, 69], followed by two Android specific techniques [48, 73].

A side-effect of modern microchip's complexity is the containment of errors. Needless to say that bugs are not transfered into the implementations of emulators. Thus, they can be used to discover the emulated environment. Furthermore, real devices have machine-dependent registers for meta states. Those are not commonly implemented for emulators either. One of the general drawbacks of emulators is its performance. The absolute execution time is significantly higher than on comparable hardware. Hence, it can also be facilitated to recognize system emulation, although it has to be stated that it is an unreliable method.

Another possible way to distinguish between an emulator and a real smartphone is to retrieve and compare hardware specific values. Emulated hardware components such

[1] **Vanilla Android** is the default android OS without any device manufacturer or provider related modifications.

as microchips, controllers, or graphic cards, as well as real hardware, contains such identifier strings. Notwithstanding the fact that these identifiers are emulator related, we discuss them in the next section because the accessibility is guaranteed by high level OS APIs.

Matenaar and Schulz [48] introduced a technique to detect if an application is executed in a QEMU ARM emulator on the x86 host system. Therefore they take advantage of the binary translation optimization of QEMU. During execution, sequences of instructions without jumps or branches are translated into x86 instruction blocks. While executed on the underlaying real hardware, there is no possibility for the simulated QEMU Central Processing Unit (CPU) to interrupt such a block. For that reason, emulated processor interrupts can only occur in between atomic blocks and as a consequence some registers such as the program counter are not restored correctly. In order to demonstrate the practicality Matenaar and Schulz implemented a proof of concept[2].

Another method presented by Schulz [73], bases on unequal caching concepts. The ARM architecture is designed with a dedicated data and instruction Level-1 (L1) cache which is not synchronized. If a value at a certain address in one cache is updated it is not necessarily updated within the other simultaneously. In contrast, the x86 architecture has only one L1 cache for both. The QEMU emulator does not consider this kind of caching peculiarity. It is exploited by executing a specific instruction. Hence, it forces the CPU to cache it. Next, data at the given address is modified. If the same instruction is re-executed on a real device it yields to the same result as before, whereat the emulator provides a result corresponding to the altered data. Note that it is a probabilistic approach, thus it does not work invariably.

For countering *Emulation Related Detection* we present conversions in Section 4.3. In the following, detection strategies which focus on the environment layer are given.

4.2.2 Environment Related Detection

Environment Related Detection denotes the exposure of OS characteristics or associated properties, which can be exploited to identify sandboxes.

This taxonomy group is particularly important due to the already outlined higher context awareness of mobile devices. Concrete techniques for the Android platform are

[2] **Proof of concept** for binary translation detection on ARM `www.dexlabs.org/files/detect_bt_arm.c`.

discussed consecutively. Note that we do not list all found prospects for identification, but rather select a representative for each class. For a full list please refer to Appendix B.
To obtain the information which possibly indicate the presence of a sandbox environment, we developed an Android application as part of this Thesis. It queries the data from two Samsung Nexus devices and additionally from the DroidBox version 4.1. The values and properties of DroidBox are compared to the smartphone data. All significant deviations are further investigated to be consistent and resilient.

As aforesaid, real hardware contains device specific identifier strings. Moreover, Android provides an entire class which serves as interface to fetch those static values. Such values include hardware strings for cards and boards, manufacturer's names, and numeric version identifiers. Additionally, values which are utilized to identify the build environment like the build time, the version, and tags can be viewed. An overview of representative identifier strings is given in Table 4.1.
In case of non-existent device identifiers the emulation software returns `null` values. The OS API handles this situation by translating them into predefined strings such as *unknown* or *generic*. The build constants are automatically gathered by the build environment.

Identifier Value	**DroidBox 4.1**	**Nexus 5**	**Nexus S**
`HARDWARE`	goldfish	hammerhead	herring
`ID`	MASTER	KOT49H	JZO54K
`MANUFACTURER`	unknown	LGE	samsung
`MODEL`	sdk	Nexus 5	Nexus S
`TAGS`	test-keys	release-keys	release-keys
`TIME`	1349813031000	1386201169000	1349214417000
`VERSION`	495790	937116	485486

TABLE 4.1: Constants accessible through the `Build` interface.

Android contains a dedicated storage space (`Settings.Secure`) for preferences that applications can read but are not allowed to alter directly. The only way to modify these settings is through the system UI or specialized APIs. The `ADB_ENABLED` value indicates whether the device is connected to the ADB or not. The emulator always returns `null`. The key `DEFAULT_INPUT_METHOD` reveals the default keyboard. Table 4.2 shows a brief synopsis of similar values.

The `System.Settings` interface contains miscellaneous system preferences (cf. Table 4.3). In contrast, these values are alterable by apps. As an illustration we choose `NOTIFICATION_SOUND` and `RINGTONE` that define the file paths for alarm tones. The

Settings Value	DroidBox 4.1	Nexus 5	Nexus S
ADB_ENABLED	null	0	0
ANDROID_ID	aa5d2dfec7725811	acb568bc17a9a502	20c141a489fa6847
DEFAULT_INPUT_METHOD	[...]latin/.LatinIME	[...]latin.LatinIME	[...]latin.LatinIME
WIFI_WATCH[...]LIST	null	GoogleGuest	GoogleGuest

TABLE 4.2: Settings accessible through the `Settings.Secure` interface.

default values for DroidBox are `null`. Therefore, it is always different compared to a real device, no matter which sound is explicitly chosen by an user. The same holds for the other two values depicted in Table 4.3. Generally, only values are considered which deviate independently from the user's concrete settings.

Settings Value	DroidBox 4.1	Nexus 5	Nexus S
NOTIFICATION_SOUND	null	content://[*path*]	content://[*path*]
RINGTONE	null	content://[*path*]	content://[*path*]
WAIT_FOR_DEBUGGER	null	0	1
WIFI_SLEEP_POLICY	null	2	2

TABLE 4.3: Settings accessible through the `Settings.System` interface.

In order to communicate with cellular networks mobile phones are equipped with Subscriber Identity Module (SIM) cards. The integrated circuit stores the IMEI, IMSI, and authentication keys. The `TelephonyManager` serves as interface and enables other applications to query these information. Apps can also register listeners for receiving notifications of telephony state changes. A selection of methods and their return values are listed in Table 4.4.

DroidBox replaced the emulator's default IMEI, IMSI, and phone number. However, the deployed values are well-known and can be preyed to identify the sandbox. The same applies to the network operator name or its identifier especially because the default values clearly state the presence of the Android emulator.

Mobile networks manage their communication services through fixed base stations. Since mobile phones communicate with a distinct station, mostly the geographically closest, they are aware of its unique cell identifier and the cell location. Whether the device location changes considerably, registered listeners are notified. The Android emulator generally does not return any cell location value except of `null`. From this constellation two detection approaches can be derived: The absence of cell location information (`getCellLocation()`) reveals DroidBox and a state not changing over time indicates a sandboxed environment.

Method	DroidBox 4.1	Nexus 5	Nexus S
`getCellLocation()`	null	[21474647,...,280]	[51509,...,-1]
`getDeviceId()`	000[...]000	358[...]793	355[...]913
`getDeviceSoftwareVersion()`	null	05	01
`getLine1Number()`	15555215554	+4917688778911	+4917688778911
`getNetworkOperatorName()`	Android	o2 - de	o2 - de
`getSimOperator()`	310260	26207	26207
`getSimSerialNumber()`	890[...]720	894[...]507	894[...]413

TABLE 4.4: Values accessible through methods of `TelephonyManager`.

As most other computer devices Android manages a range of timing values. The class `SystemClock` offers methods to obtain time related values which can be leveraged to determine if the execution environment is emulated or not. The by `uptimeMillis()` returned long value is counted in milliseconds since the system was booted and excludes device inactivity. Hence, the time periods whereat the CPU is off, the display is dark, or the device waits for external input is subtracted from the total uptime. In contrast, `elapsedRealtime()` gives access to the time since the system was booted including deep sleep. In practice, smartphones are not regularly turned off and usually run for days. Based on the following three statements the detection of a sandbox may be performed:

- *The uptime is very low.* Most sandboxes run only a few minutes until they are reset. According to our survey a threshold of 10 minutes leads to meaningful results.
- *The values of `uptimeMillis()` and `elapsedRealtime()` are inconsistent.* For example the value returned by `elapsedRealtime()` is the lower one or any of the two is zero.
- *The difference of `uptimeMillis()` and `elapsedRealtime()` is not significant.* Mostly a smartphone is switched on the entire day but is only actively used temporarily. Thus, a similar value strongly indicates that the code is not executed onto a real device.

All major Android device manufactures ship their devices with a preinstalled Play Store application. Neither DroidBox, nor the Android emulator is featured with the Play Store or any other Google application. Owing to technical restrictions it is even impeded to install it belated. As recently disclosed, Google contracts obligate them to install all Google apps, even if they just want a single application such as the Play Store or Google Maps [17]. Thereby, the tie contracts render the opportunity to analyze the installed

Google apps for our purposes.
The absence of the Play Store is considered as an evidence for an artificial environment. In addition, a list of all installed apps can easily be retrieved and compared with an enumeration of expected Google apps. Eventually, the available software versions and signatures are verified and tested for anomalies.

The steady changes of geo locations of smartphones exhibit more distinctive features compared to DroidBox. Each device maintains a list of ever connected WLAN access points. Obviously, the list of the emulator and thence DroidBox is empty by default. Another notable characteristic of real devices is the continuous alteration of the wireless signal strength or battery level. DroidBox always possesses the absolute identical merits.

We conclude that a huge amount of *Environment Related Detection* features could be discovered and implemented. The following subsection focuses on the exploiting of a lack of user driven events which permanently occur on real mobile devices.

4.2.3 User Input Related Detection

The users of smartphones permanently interact with their devices. This triggers numerous events and a variety of data is created, updated, retrieved, and deleted. Unusual or missing state changes can be traced and advantaged to unfold sandboxes. The taxonomy's class *User Input Related Detection* unites such detection measures.

An outstanding asset of modern mobile OSs is the capability to install apps throughout centralized application market places. Users commonly take advantage of this feature and download numerous apps and games. Hence, the absence of third party apps seems on account of that unusual. Android generally allows to obtain a full enumeration of installed apps without dedicated permission. Simple rule based detection, such as *"All installed app's package names conform the namespace* `com.android.*`*"* is encountered as effective already.
Not only missing apps are a strong indicator for the app execution within a sandbox, also missing user-data is believed to be suspicious. Due to the user's interaction with various apps a large amount of contextual data is generated, including system as well as third-party apps. The address book has to contain contacts, the list of outgoing or incoming calls may not be empty, and the SMS app shall hold received and send messages. Third-party applications such as Twitter [91] or WhatsApp [97] cannot be searched directly for communication data but state changes are announced by the global

notifications API. In this way new appearing notifications can be monitored.
Finally, even the presence of user-data must not necessarily lead to the conclusion that an app runs on a real smartphone. For instance, outdated e-mail messages could be considered odd. Since apps commonly store data within databases the last modified timestamp can be checked.

4.2.4 Limitations

The developed taxonomy is by no means a complete listing of all existing Android sandbox detection approaches. As already clarified in Section 1.3, we omit rootkits since they have unlimited access to all OS components and resources. Due to their nature, there is no need for the application of the intended OS APIs. Thus, effective detection mitigation (cf. Section 4.3) cannot be applied in this case.

4.3 Sandbox Detection Mitigation

As already demonstrated within the preceding sections, several detection techniques which address different properties of sandboxed environments could be contrived. The goal of this monograph is to elaborate effective detection mitigation based on the introduced taxonomy. Therefore, we seek to revise DroidBox in a manner that it is indistinguishable from real devices.
Consequently, we seize our taxonomy and propose associated evasion measures to counter detection, followed by the in practice applied extension of DroidBox. To begin with, we examine countermeasures for *Emulator Related Detection* briefly. Afterwards, *Environment Related Detection* mitigation is covered more in depth. We close the discussion with some final thoughts on how to encounter *User Input Related Detection*.

4.3.1 Emulator Related Detection Mitigation

When considering *Emulator Related Detection* the underlaying issue of virtualization comes into play. VMMs and emulators are not designed with malware analysis and detection mitigation in mind. For general purposes it is enough to be adequate close to real systems but there is no need to behave absolute identical. Thus, it is theoretical feasible to remedy each emulation detection approach by adapting the system accordingly. Admittedly this is too costly on a large scale.

For evading the exposure of DroidBox's system emulation profound changes have to be accomplished. Since the first explained concept targets binary translation and the other ARM's caching architecture, QEMU's core design needs to be altered. Unfortunately, this is not the primary goal of the emulator developers because it is actually utilized to support development and testing of Android applications. However, two procedures remain for hardening the emulated environment: First, the sandbox could be compiled and deployed on the x86 architecture which is generally supported by Android but not by TaintDroid. Secondly, hardware assisted acceleration such as implemented for x86 has already been announced for ARM and enables analysis environments to be deployed on native hardware [3].
Nevertheless, circumventing emulation or virtualization detection in general remains a subject of ongoing research and is considered for future work in Section 6.2.

4.3.2 Environment Related Detection Mitigation

This passage groups all anti-detection measures to thwart identification of environment related properties. Therefore, we do not only suggest theoretical ideas but implement our findings in DroidBox. To this end, the in Chapter 3 examined and faultless attested operation of DroidBox in version 4.1 serves as basis. Since the OS itself just as the DroidBox add-on, is publicly available it appears easily expandable. In oppose to OSs compiled for specific devices our implementation targets the more flexible Android emulator. Thus, we do not have to satisfy hardware or driver related constraints. In order to avoid difficulties with generic emulator drivers our advancements are performed on the two upper layers of the Android architecture (cf. Section 2.1).

The device identifiers which are leveraged for detection are necessarily traceable within the environment's software. Thus, it is applicable to either change the values at the place whereat they are stored or the concerning API to retrieve it. For the DroidBox we modified the internal API as illustrated in Listing 4.1.

```
private static String getString(String property) {
        if (property.equals("ro.bootloader"))
                return "I9020XXLC2";
        [...]
        return SystemProperties.get(property, UNKNOWN);
}
```

LISTING 4.1: Interception of `getString()` calls for certain strings.

Within the `getString()` method the intended control flow is intercepted if the requested string is of interest. With several sequential if statements, values identical to the Nexus S are returned. Note that we could not simply alter the constant strings in `SystemProperties` because the Android build environment checks external APIs and constants for modification and aborts the build in case of deviations.

The procedure for patching the Android preferences is analogous to the device identifiers. Values are stored in the system and can be retrieved by an API. The settings databases might be adjusted at any startup again. Thus, the modification of the API is more generic and therefore preferred (cf. Listing 4.2). A lot of work needs to be spent on adjusting all single setting related detection features. The settings API solely provides a centralized interface and delegates the request to subclasses.

```
public synchronized static String getString(ContentResolver resolver,String name){
        if (name.equals(Secure.ADB_ENABLED))
                return "0";
        if (name.equals(Secure.WIFI_WATCHDOG_WATCH_LIST))
                return "GoogleGuest";
        [...]
```

LISTING 4.2: Interception of getString() calls for certain settings.

The `TelephonyManager` serves as interface to query cellular network relevant information. Data is not accessed key-based using a single method but trough a set of special purpose functions. On account of this, all traitorous methods are changed and return values equal to a Nexus S. Particularly, the adjustments of the `Exception` handling and the transition of the taint sources have to be carried out carefully.
An example in Listing 4.3 demonstrates the modification of the `getSimSerialNumber()` method which returns the SIM serial number. For that number as well as for IMEI and IMSI the replacement value has to be consistent with validation criteria and satisfy a standardized format.
The cell location which is retrievable via `getCellLocation()` passes the location value as an array. Unlike the static identifiers or settings, these values depend on the current geo location on real hardware. Thus, it is not a static but rather continuous. In order to thwart blacklisting predefined locations a random value in a realistic range is employed. All other methods of the `TelephonyManager` are adapted in a similar manner.

```
public String getSimSerialNumber() {
        String simSerialNumber = "8949228121903158413";
```

```
        try {
                Taint.addTaintString(simSerialNumber, Taint.TAINT_ICCID);
                return simSerialNumber;
        } catch (NullPointerException ex) {
                return null;
        }
}
```

LISTING 4.3: Anti-detection adjustment for the SIM serial number.

The detection based on the uptime is eliminated by the extension of `SystemClock`. We define an initialization for `elapsedRealtime()` value which varies slightly and express a timespan of approximately one day whereby uptimeMillis() is initialized with 30 minutes. Hence, non of the three statements to encounter the presence of a sandbox based on timing is satisfied (cf. Subsection 4.2.2).

According to Google, the Play Services Framework including the Play Store cannot be installed on the emulator and thereat not on DroidBox either. For testing purpose on the Android emulator they provide a Play Services SDK as plugin for common IDEs. However, it is still feasible to get Google apps and the Play Store running on DroidBox. Therefore, ADB is utilized to download the needed APK files from a real device. The inevitably required files, *GoogleLoginService.apk*, *GoogleServicesFramework.apk*, and *Phonesky.apk* are stored at */system/app/* on each physical Android devices. The retrieved files have to be copied in the same folder on the user-data image file. When the DroidBox system is booted with the modified user-data image file the Play Store is present and can be used to download other apps. Accordingly, any proprietary Google App can be installed into DroidBox.

The frequently changing signal strength and a decreasing battery level is simulated by the standard emulator framework. It allows to alter the concerned values with automated scripts from outside the emulator instance. The same holds for geo positions. The Android SDK contains a program which generates synthetic location data for the device. It only needs to be set up properly and does not require any further modification.

In essence, all introduced environment related detection measures of Section 4.2 could be annulled theoretically. For this reason we evaluate the practical value of our findings in Chapter 5. The presented additions and improvements can be found as patch file on the enclosed DVD.

4.3.3 User Input Related Detection Mitigation

In order to mitigate detection through the absence of user-triggered events or a lack of user-data both conditions must be simulated. Thus the goal of this subsection is to equip DroidBox with realistic user-data.

Installing a certain number third-party apps circumvents the uncovering of sandbox environments. The detection vector, enabled by missing third-party applications, is addressable by simply installing an arbitrary amount of programs. Google's Play store devotes a dedicated interface for communicating with Android devices and enables them to download apps. It is facilitated in the not intended manner to download application packages, which in terms can be transfered onto DroidBox. Therefore ADB is employed, as well as for the final installation. This entails the advantage that the entire process of downloading and deploying can be automated and repeated on a daily basis. Thus, even the app versions are in a large part up-to-date.

In order to not appear suspicious all installed apps are filled with user-data. For the Google powered apps such as Mail, Contacts, Keep, and Calendar the active sync technology is applied. It syncs all enumerated apps continuously. For the other applications, it is not quite as simple since they have to be filled manually with consistent data. Calls or incoming messages are events which can also be triggered by the Android SDK during an analysis. The applied modifications are included in the user-data image of DroidBox which can be reset to the initial state before each analysis. Especially the desire to automated the time consuming procedure is only prototypical implemented yet.

User Input Related Detection is, in contrast to the taxonomy's two other groups, not as precise. The accruement of user-data is a very complex process and an accurate detection based on such information is challenging. In a practical use case the number of false positives would be unjustifiable high. For this reason we doubt a large-scale deployment in the near future. On the other hand, it will always remain at least a probabilistic detection approach because the imitation in order to mitigate detection is exceedingly complicated.

4.4 Summary

Considering the huge amount of malware that strongly facilitates code obfuscation techniques to prohibit static analysis already, it becomes clear that a vast number of samples

will employ techniques to evade dynamic analysis in the near future.
This chapter aimed at structuring methods to reveal sandbox environments for Android and introduced a taxonomy of detection vectors. Hence, we clustered measures in three groups: *Emulator Related Detection*, *Environment Related Detection* and *User Input Related Detection*. According to this taxonomy we evidenced with concrete implementations that there are plenty of properties which can be exploited to determine the presence of DroidBox. Our findings are evaluated by means of web-based analysis systems in Chapter 5.

In addition, we feature anti-detection measures aligned with the developed taxonomy. The advancements were implemented on the basis of the discussed version of DroidBox in Chapter 3. Thereby, all introduced *Environment Related* and *User Input Related* detection measures were warded off successfully. Finally, the hardened DroidBox is integrated in the Mobile Sandbox analysis system and evaluated in the next Chapter.

Clearly, not only a wide range of practical sandbox detection measures were found and categorized, but also countermeasures implemented for DroidBox could be put in place.

Chapter 5

Evaluation

After describing sandbox detection techniques, followed by mitigation approaches and their implementation in DroidBox (cf. Chapter 4), we finally evaluate our contributions. Therefore, the identified detection vectors are investigated by means of online sandbox systems and our extension of DroidBox, which is integrated into the Mobile-Sandbox NG. This evaluation will be discussed in the upcoming sections, the proceeding will be as follows: At first, the test setting is explained and the results are illustrated in Section 5.1. Afterwards, we briefly regard the integration of the extended DroidBox into the Mobile-Sandbox NG. Finally the chapter closes with a short summary.

5.1 Testing: Detection and Mitigation

In Section 4.2, we demonstrated a variety of methods that uncover dynamic malware analysis environments for Android. Thus, we seek to demonstrate the effectiveness of our findings within this monograph.

Security vendors and researchers deployed several dynamic analysis environments for Android applications allowing submissions via a web-interface. Thus, even inexperienced users can verify with little effort if apps are harmless. In addition, the systems serve as source for yet unknown samples. We used a set of such online sandboxes to evaluate our detection techniques in a real-world scenario.
Our testing environment consists of an Android application and a server connected to the Internet. The app is uploaded to a sandbox web-service and upon execution the required data is retrieved. The app comes with a GUI and solely sends the queried

information through a Transmission Control Protocol (TCP) socket to the server. The software running at the server is build to receive and store the data into a log file (cf. Appendix B). Each submitted sample contains an unique identifier to be related to a distinct service and in order to track redistribution. The source code for both programs can be found on the attached DVD (cf. Appendix D).

In total, we found 14 online sandboxes for Android (cf. Appendix C). Upon closer examination, however, eight were not longer considered: *App-Ray*, *Aurasium*, *CopperDroid*, and *Sandroid* were not available anymore or outdated and thus not compatible with our app. Other analysis services did not send any results to our server or do not perform a dynamic analysis, namely *AndroTotal* and *VisualThreat*. The two app stores, *Google Play* and *Yandex Store*, were expected to perform a dynamic analysis as security measure to prevent malware from redistribution, but we did not receive any data. Additionally, we noticed that some sandboxes forward the provided app to other unidentified sandbox systems. Those are excluded from our evaluation.
Eventually, we analyze a set of six Android sandboxes by means of the following features (or functions) which are explained in Section 4.2 extensively:

- *Identifier Strings* - Considers a set of identifier strings, for instance `BOOT`, `BRAND`, `DEVICE`, and `DISPLAY`. Only if all strings are unsuspicious the detection has failed.
- *ADB Enable* - Most sandbox emulators communicate with their environment through ADB. This flag indicates the current connectivity status.
- *Default Input Method* - Interprets the package name of the default on-screen keyboard.
- *WiFi Watch List* - A list of WLAN access points. This value is on real devices not `null`.
- *Sounds* - Evaluates if for the alarm, messages, and call notification sounds are defined.
- *Cell Locations* - Validates the cell location, must not be `null` on physical devices.
- *Device ID* - Checks if the IMEI is different from DroidBox or not 0000000000.
- *Phone Number* - Checks if the phone number is different from DroidBox and the emulator.
- *Uptime* - Approves the in Subsection 4.2.2 defined rules for the uptime.

- *Play Store* - Determines whether the Play Store is installed or not.
- *Third Party Apps* - Searches for third-party applications such as Twitter or WhatsApp.
- *Messages* - Reads email and SMS messages.

The results are summarized in Table 5.1. If one of the above described detection vectors could identify a sandbox with certainty it is denoted as ✓. In case the outcome is not explicit or identical with real devices it is labeled with ✗.

Sandbox	Identifier Strings	ADB Enabled	Default Input Method	WiFi Watch List	Sounds	Cell Location	Device ID	Line Number	Uptime	Play Store	Third Party Apps	Messages
Andrubis	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✓
ApkScan	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓	✓
Foresafe	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
JoeSandbox	✓	✓	✗	✓	✗	✓	✗	✗	✓	✓	✓	✗
TraceDroid	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓
Mobile-Sandbox	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓
Mobile-Sandbox NG	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗
DroidBox 4.1	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

TABLE 5.1: List of evaluated sandboxes and detection vectors (Legend: Identified the sandbox = ✓, not or not clearly identified = ✗).

The overview depicts clearly that our detection methods work remarkably for the evaluated sandboxes. However, some features are countered successfully. The most sophisticated camouflage is established by *JoeSandbox*, but most systems fail to even conceal the most trivial identification properties. We conclude that all tested sandbox systems can successfully be detected with absolute certainty, in a fraction of a second with only few lines of code. Even if the vendors act on our suggestions we are confident to still determine the presence of their systems at least by combining a set of features.

In Section 4.3, we introduced countermeasures to prevent a detection by evasive malware. The verified version of DroidBox is extended by these anti-detection techniques. Afterwards, it is integrated into the online analysis environment Mobile-Sandbox. Table 5.1 enlists two different versions, the term *Mobile-Sandbox* denotes the old version which utilized a customized DroidBox 2.3 as basis, whereat *Mobile-Sandbox NG (Next*

Generation) refers to an updated facility.
The primarily Mobile-Sandbox is prone to be detected by malware, merely the IMEI and the phone number are altered by default. In contrast, the Mobile-Sandbox NG overcomes these limitations and is featured with our extensions to ward even sophisticated malware off. Details on the integration are outlined in the following section.

5.2 Mobile-Sandbox Integration

Within this paragraph we briefly introduce the Mobile-Sandbox (cf. Figure 5.1), followed by a reflection of the the integration process of the extended DroidBox into the environment.

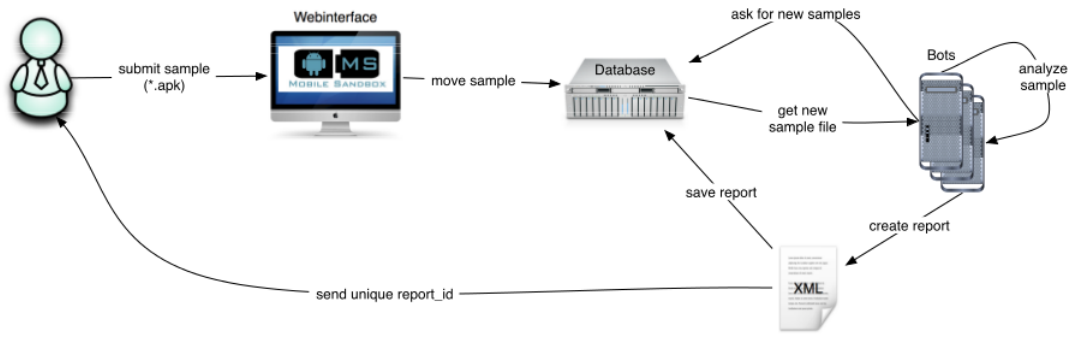


FIGURE 5.1: Mobile-Sandbox workflow [79].

The Mobile-Sandbox consists of a web-interface, a database server, and an emulated DroidBox to perform the analysis. The user or analyst initiates the workflow by submitting a sample. Prior to the analysis, meta data is extracted and the APK file is stored on the database server. A corresponding unique identifier is added to a queue, which supplies the DroidBox analysis bots with new samples. After completion of the examination, a XML report is generated and stored in the database. The initiator is notified about the result and receives the generated unique identifier to demand the report. Since the Mobile-Sandbox utilizes the Android emulator running DroidBox 2.3 we could simply replace the emulator images for integration. Furthermore, the system is equipped with realistic user-data by adding a prepared user-data image file. Upon each performed analysis the environment is reset to provide an equal, well defined environment for every examination.
At the time of writing the updated Mobile-Sandbox NG analyzed already about 5,000

uploaded malware samples. Random analysis results were inspected manually and compared to reports of other sandbox systems on the Internet. The resulting action sequences correlated to behaviors that were described in the other analysis reports. These insights gave us confidence that the DroidBox extension is working correctly.

5.3 Summary

This chapter aimed at evaluating the proposed sandbox detection methods and the effectiveness of the elaborate countermeasures. For this, an app was developed and submitted to several online sandbox environments. They could be clearly flagged as analysis systems, unless we put our anti-detection strategy in place. We improved the Mobile-Sandbox in a manner that it overcomes all proposed detection strategies.

Chapter 6

Conclusion

This final chapter closes the Thesis with a summary and discussion in Section 6.1, which also outlines our contributions. Finally, Section 6.2 elaborates on possible future work.

6.1 Summary and Discussion

At the end of Section 1.2 we summarized the objectives of this work as follows: *The goal is to build an Android sandbox that is notably harder to detect by malware in comparison to DroidBox. Additionally, it is supposed to be compatible with up-to-date Android apps.* Thus, we basically aim to answer two not directly related key questions:

First, we attempt to eliminate the shortcoming of the DroidBox version 2.3. Since it is based on an outdated version of Android only a subset of existing applications can be analyzed. To this end, the obvious solution is to port DroidBox to a more recent version of Android. In order to achieve this goal we do not extend the original Android OS but leverage the Android extending taint tracking system TaintDroid. Due to the fact that during our work it emerged that a third-party ported version of DroidBox 4.1 occurred, we supplementary verify its accurate operation. Therefore, we developed a test strategy that proves the correctness by applying three independent test scenarios, briefly exemplified in the following.
The Android build tests are adducted to demonstrate the absence of critical errors. Besides, we develop an automated framework to compare results of dynamic malware analyses between DroidBox 2.3 and 4.1. The last test setting employs malware with known malicious behavior to examine defined functionality of DroidBox. The outcomes

of all tests are positive, thus we prove DroidBox 4.1 is fully functional. Moreover, the established and conducted testing strategy is universally valid and can also be engaged by future ports.
In principle, porting DroidBox to a newer version of Android is a recurring task which has to be fulfilled with each update. To aid this process by means of automation, we port DroidBox 4.1 to Android 4.4 and aspire to identify a generalized approach. The porting itself is a notably more complex task than extending TaintDroid 4.1 to DroidBox 4.1, which originally was planned (cf. Section 1.2) because the TaintDroid modifications must be considered as well. Thereby, we developed a toolchain to semi-automate the process of identifying the modified files for further manual investigation. However, we conclude that a fully automated environment for porting DroidBox to further Android versions cannot be applied with reasonable effort. The complexity of an OS in combination with the sophisticated taint tracking system requires expert judgment of an experienced developer, which can hardly be transformed into a model.

Secondly, we address defense strategies employed by malware to thwart analysis by sandboxes. Code obfuscation is applied in order to prohibit static code analysis, whereas sandbox detection seeks to circumvent automated dynamic analysis. If an instance becomes aware of being executed within an analysis environment it may adapt its behavior to appear benign. Unfortunately, this prevents conclusions on the threat and an estimations on the impact effectively.
Our work is the first to present a meaningful taxonomy which clusters sandbox detection measures into three groups: *Emulator Related Detection*, *Environment Related Detection*, and *User Input Related Detection*. Accordingly, we developed a set of reliable detection features targeting the discussed version of DroidBox. Within the evaluation (cf. Chapter 5) the identified groups are appraised by means of different web-based analysis environments.
Besides the wide range of practical sandbox evasion techniques found and categorized, countermeasures were also established. We feature anti-detection measures in alignment with the developed taxonomy and successfully combat all introduced approaches. From malware's point of view, our extension of DroidBox is indistinguishable from a real device.

Our work was not solely a theoretical experiment, but in addition integrated into the real world solution Mobile-Sandbox. As the basis for the web-based analysis system, it was used to analyze over 5,000 samples at the time of writing. Furthermore, we state that our findings can be applied to a wide range of Android sandboxes since we are going

to give public access to the research community.
We believe our work makes important contributions to assist analysts in combating the emerging threats of evasive mobile malware through an improved version of DroidBox and putting the defenders a step ahead. Moreover, we are aware of the ongoing arms race. Yet, to our knowledge there are no distinct detection features. Other approaches relying mainly on probabilistic methods that lead to a significant number of false positives. This is clearly not in the interest of attackers because they aim to affect as many devices as possible. Thus, we finally claim anti-detection needs not to be perfect. It is enough to raise the bar sufficiently high so that attackers run the risk of omitting too many real devices and thereby loose profit.

6.2 Future Work

The field of dynamic analysis of mobile malware was not subject to extensive research yet. Especially sandbox detection and mitigation strategies have been omitted. Based on the conducted results within this Thesis, we propose to initiate the following subjects for future research:

- As soon as the DVM becomes obsolete, a new taint tracking system for the ART is required. It is advisable to implement it in a more generic way to alleviate future porting intentions.
- To date, neither DroidBox, nor any other taint tracking system is capable of tracking implicit flows [13, 30, 72]. This is a significant drawback and should be considered for future research.
- As stated in Chapter 3, the extensions of DroidBox are not adequately labeled within the source code. It complicates porting unnecessarily and can be avoided by low expenditure on subsequent adjustments.
- The question of emulator related detection mitigation remains largely unanswered. Thus, an investigation of this field of study is desirable.
- Dynamic analysis detection is employed by malware to adjust the behavior according to the outcome. Thus, the program code, which is required to expose sandboxes, is available inevitably. It seems to be a promising approach that may yield to the determination of sandbox detection code by means of static analysis.

- There is a strong need for a procedure that challenges emulator detection in general. An auspicious approach, which seems to eliminate most outlined environment related detection methods and all emulation-based techniques, is to deploy the analysis environment onto a real device. Hence, the feasibility is still completely open.

- Finally, an important question is how to generate meaningful reports to summarize analysis results. This is a general issue of IT security and defined as *semantic gap* by Sommer and Paxson [77].

Appendix A

Android Build Instruction Manual

Build instructions for Android on Ubuntu x64 12.04 LTS

Install Java

- `sudo apt-get purge openjdk*`
- `sudo add-apt-repository ppa:webupd8team/java`
- `sudo apt-get update`
- `sudo apt-get install oracle-java6-installer`

Install libs and stuff

- `sudo apt-get install git gnupg flex bison gperf build-essential zip \`
  `curl libc6-dev libncurses5-dev:i386 x11proto-core-dev libx11-dev \`
  `:i386 libreadline6-dev:i386 libgl1-mesa-glx:i386 libgl1-mesa-dev \`
  `g++-multilib mingw32 tofrodos python-markdown libxml2-utils \`
  `xsltproc zlib1g-dev:i386`
- `sudo ln -s /usr/lib/i386-linux-gnu/mesa/libGL.so.1 /usr/ \`
  `lib/i386-linux-gnu/libGL.so`

Install curl

- `sudo apt-get install curl`

Install and setup repo-tool

- `mkdir  /bin`
- `PATH= /bin:$PATH`
- `curl http://commondatastorage.googleapis.com/git-repo-downloads \ /repo >  /bin/repo`
- `chmod a+x  /bin/repo`

Download the source

- `mkdir WORKING_DIRECTORY`
- `cd WORKING_DIRECTORY`
- `repo init -u https://android.googlesource.com/platform/manifest -b \ android-4.1.1_r6`
- `repo sync`

Prepare the build env

- `source build/envsetup.sh`
- `lunch` //and choose traget (interactive)

Build

- `make -jX` //(whereat X defines the number of processes)
- **Good luck!**

Appendix B

List of Detection App Output

All evaluated detection features received by our server, sent from our app while it is analysed within an online sandbox.

```
Wed Mar 05 10:26:10 CET 2014 Waiting for a new Client...
Wed Mar 05 10:26:10 CET 2014 /128.68.25.241:56540 has joined.
 ID_19241_13[...]22 > ID_19241_1388187349922
 ID_19241_13[...]22 > TelephonyManager.getCallState(): 0
 ID_19241_13[...]22 > TelephonyManager.getCellLocation(): null
 ID_19241_13[...]22 > TelephonyManager.getDataActivity(): 0
 ID_19241_13[...]22 > TelephonyManager.getDataState(): 2
 ID_19241_13[...]22 > TelephonyManager.getDeviceId(): 354314058663839
 ID_19241_13[...]22 > TelephonyManager.getDeviceSoftwareVersion(): 78
 ID_19241_13[...]22 > TelephonyManager.getLine1Number(): +79264567184
 ID_19241_13[...]22 > TelephonyManager.getNeighboringCellInfo(): []
 ID_19241_13[...]22 > TelephonyManager.getNetworkCountryIso(): ru
 ID_19241_13[...]22 > TelephonyManager.getNetworkOperator(): 25002
 ID_19241_13[...]22 > TelephonyManager.getNetworkOperatorName(): Beeline
 ID_19241_13[...]22 > TelephonyManager.getNetworkType(): 2
 ID_19241_13[...]22 > TelephonyManager.getPhoneType(): 1
 ID_19241_13[...]22 > TelephonyManager.getSimCountryIso(): ru
 ID_19241_13[...]22 > TelephonyManager.getSimOperator(): 25001
 ID_19241_13[...]22 > TelephonyManager.getSimOperatorName(): MTS
 ID_19241_13[...]22 > TelephonyManager.getSimSerialNumber(): 8940195201326570
 ID_19241_13[...]22 > TelephonyManager.getSimState(): 5
 ID_19241_13[...]22 > TelephonyManager.getSubscriberId(): 250017103105458
 ID_19241_13[...]22 > TelephonyManager.getVoiceMailAlphaTag(): Voicemail
 ID_19241_13[...]22 > TelephonyManager.getVoiceMailNumber(): initLog0600
 ID_19241_13[...]22 > TelephonyManager.hasIccCard(): true
 ID_19241_13[...]22 > TelephonyManager.isNetworkRoaming(): false
 ID_19241_13[...]22 > Build.BOARD: smdk4x12
 ID_19241_13[...]22 > Build.BOOT: unknown
 ID_19241_13[...]22 > Build.BRAND: samsung
```

```
ID_19241_13[...]22 > Build.CPU_ABI: armeabi-v7a
ID_19241_13[...]22 > Build.CPU_ABI2: armeabi
ID_19241_13[...]22 > Build.DEVICE: m0
ID_19241_13[...]22 > Build.DISPLAY: JZO54K.I9300XXEMG4
ID_19241_13[...]22 > Build.FINGERPRINT: samsung/m0xx/[...]:user/release-keys
ID_19241_13[...]22 > Build.HARDWARE: goldfish
ID_19241_13[...]22 > Build.HOST: SEP-108
ID_19241_13[...]22 > Build.ID: JZO54
ID_19241_13[...]22 > Build.MANUFACTURER: samsung
ID_19241_13[...]22 > Build.MODEL: GT-I9300
ID_19241_13[...]22 > Build.PRODUCT: m0xx
ID_19241_13[...]22 > Build.RADIO: unknown
ID_19241_13[...]22 > Build.SERIAL: unknown
ID_19241_13[...]22 > Build.TAGS: release-keys
ID_19241_13[...]22 > Build.TIME: 1373974353000
ID_19241_13[...]22 > Build.TYPE: user
ID_19241_13[...]22 > Build.USER: se.infra
ID_19241_13[...]22 > Build.getRadioVersion():
ID_19241_13[...]22 > Build.VERSION.CODENAME: REL
ID_19241_13[...]22 > Build.VERSION.INCREMENTAL: I9300XXEMG4
ID_19241_13[...]22 > Build.VERSION.RELEASE: 4.1.2
ID_19241_13[...]22 > Build.VERSION.SDK: 16
ID_19241_13[...]22 > Build.VERSION.SDK_INT: 16
ID_19241_13[...]22 > Debug.isDebuggerConnected: false
ID_19241_13[...]22 > Secure.ACCESSIBILITY_ENABLED: null
ID_19241_13[...]22 > Secure.ACCESSIBILITY_SPEAK_PASSWORD: 0
ID_19241_13[...]22 > Secure.ADB_ENABLED: null
ID_19241_13[...]22 > Secure.ALLOWED_GEOLOCATION_ORIGINS: null
ID_19241_13[...]22 > Secure.ALLOW_MOCK_LOCATION: 1
ID_19241_13[...]22 > Secure.ANDROID_ID: 6e9967cca4921743
ID_19241_13[...]22 > Secure.BLUETOOTH_ON: 0
ID_19241_13[...]22 > Secure.DATA_ROAMING: 1
ID_19241_13[...]22 > Secure.DEFAULT_INPUT_METHOD: com.android.inputmethod.[...]
ID_19241_13[...]22 > Secure.DEVELOPMENT_SETTINGS_ENABLED: null
ID_19241_13[...]22 > Secure.DEVICE_PROVISIONED: 1
ID_19241_13[...]22 > Secure.ENABLED_ACCESSIBILITY_SERVICES: null
ID_19241_13[...]22 > Secure.ENABLED_INPUT_METHODS: com.android.inputmethod.[...]
ID_19241_13[...]22 > Secure.HTTP_PROXY: null
ID_19241_13[...]22 > Secure.INPUT_METHOD_SELECTOR_VISIBILITY: null
ID_19241_13[...]22 > Secure.INSTALL_NON_MARKET_APPS: 0
ID_19241_13[...]22 > Secure.LOCATION_PROVIDERS_ALLOWED: gps
ID_19241_13[...]22 > Secure.LOCK_PATTERN_ENABLED: 0
ID_19241_13[...]22 > Secure.LOCK_PATTERN_TACTILE_FEEDBACK_ENABLED: 0
ID_19241_13[...]22 > Secure.LOCK_PATTERN_VISIBLE: 0
ID_19241_13[...]22 > Secure.NETWORK_PREFERENCE: 1
ID_19241_13[...]22 > Secure.SELECTED_INPUT_METHOD_SUBTYPE: -1
ID_19241_13[...]22 > Secure.SETTINGS_CLASSNAME: null
ID_19241_13[...]22 > Secure.SYS_PROP_SETTING_VERSION: null
ID_19241_13[...]22 > Secure.TOUCH_EXPLORATION_ENABLED: 0
ID_19241_13[...]22 > Secure.TTS_DEFAULT_PITCH: null
ID_19241_13[...]22 > TTS_DEFAULT_RATE: null
```

```
ID_19241_13[...]22 > Secure.TTS_DEFAULT_SYNTH: null
ID_19241_13[...]22 > Secure.ACCESSIBILITY_TTS_ENABLED_PLUGINS: null
ID_19241_13[...]22 > Secure.USB_MASS_STORAGE_ENABLED: 1
ID_19241_13[...]22 > Secure.USE_GOOGLE_MAIL: null
ID_19241_13[...]22 > Secure.WIFI_MAX_DHCP_RETRY_COUNT: 9
ID_19241_13[...]22 > Secure.WIFI_MOBILE_DATA_TRANS_WAKELOCK_TIMEOUT_MS: null
ID_19241_13[...]22 > Secure.WIFI_NETWORKS_AVAILABLE_NOTIFICATION_ON: 1
ID_19241_13[...]22 > Secure.WIFI_NETWORKS_AVAILABLE_REPEAT_DELAY: null
ID_19241_13[...]22 > Secure.WIFI_NUM_OPEN_NETWORKS_KEPT: null
ID_19241_13[...]22 > Secure.WIFI_ON: 0
ID_19241_13[...]22 > Secure.WIFI_WATCHDOG_ON: 1
ID_19241_13[...]22 > Secure.WIFI_WATCHDOG_WATCH_LIST: null
ID_19241_13[...]22 > System.ACCELEROMETER_ROTATION: 1
ID_19241_13[...]22 > System.AIRPLANE_MODE_ON: 0
ID_19241_13[...]22 > System.AIRPLANE_MODE_RADIOS: cell,bluetooth,wifi,nfc,wimax
ID_19241_13[...]22 > System.ALARM_ALERT: content://media/internal/audio/media/5
ID_19241_13[...]22 > System.ALWAYS_FINISH_ACTIVITIES: null
ID_19241_13[...]22 > System.ANIMATOR_DURATION_SCALE: null
ID_19241_13[...]22 > System.APPEND_FOR_LAST_AUDIBLE: null
ID_19241_13[...]22 > System.AUTO_TIME: 1
ID_19241_13[...]22 > System.AUTO_TIME_ZONE: 1
ID_19241_13[...]22 > System.BLUETOOTH_DISCOVERABILITY: null
ID_19241_13[...]22 > System.BLUETOOTH_DISCOVERABILITY_TIMEOUT: null
ID_19241_13[...]22 > System.DATE_FORMAT: null
ID_19241_13[...]22 > System.DEBUG_APP: null
ID_19241_13[...]22 > System.DIM_SCREEN: 1
ID_19241_13[...]22 > System.DTMF_TONE_WHEN_DIALING: 1
ID_19241_13[...]22 > System.END_BUTTON_BEHAVIOR: null
ID_19241_13[...]22 > System.FONT_SCALE: null
ID_19241_13[...]22 > System.HAPTIC_FEEDBACK_ENABLED: 1
ID_19241_13[...]22 > System.MODE_RINGER: 2
ID_19241_13[...]22 > System.MODE_RINGER_STREAMS_AFFECTED: 166
ID_19241_13[...]22 > System.MUTE_STREAMS_AFFECTED: 46
ID_19241_13[...]22 > System.NEXT_ALARM_FORMATTED:
ID_19241_13[...]22 > System.NOTIFICATION_SOUND: content://media/[...]/media/9
ID_19241_13[...]22 > System.RADIO_BLUETOOTH: null
ID_19241_13[...]22 > System.RADIO_CELL: null
ID_19241_13[...]22 > System.RADIO_NFC: null
ID_19241_13[...]22 > System.RADIO_WIFI: null
ID_19241_13[...]22 > System.RINGTONE: content://media/internal/audio/media/7
ID_19241_13[...]22 > System.SCREEN_BRIGHTNESS: 102
ID_19241_13[...]22 > System.SCREEN_BRIGHTNESS_MODE: 0
ID_19241_13[...]22 > System.SCREEN_OFF_TIMEOUT: 60000
ID_19241_13[...]22 > System.SETUP_WIZARD_HAS_RUN: null
ID_19241_13[...]22 > System.SHOW_GTALK_SERVICE_STATUS: null
ID_19241_13[...]22 > System.SHOW_PROCESSES: null
ID_19241_13[...]22 > System.SOUND_EFFECTS_ENABLED: 1
ID_19241_13[...]22 > System.STAY_ON_WHILE_PLUGGED_IN: 1
ID_19241_13[...]22 > System.SYS_PROP_SETTING_VERSION: null
ID_19241_13[...]22 > System.TEXT_AUTO_CAPS: null
ID_19241_13[...]22 > System.TEXT_AUTO_PUNCTUATE: null
```

```
 ID_19241_13[...]22 > System.TEXT_AUTO_REPLACE: null
 ID_19241_13[...]22 > System.TEXT_SHOW_PASSWORD: null
 ID_19241_13[...]22 > System.TIME_12_24: null
 ID_19241_13[...]22 > System.TRANSITION_ANIMATION_SCALE: 1.0
 ID_19241_13[...]22 > System.VIBRATE_ON: null
 ID_19241_13[...]22 > System.VOLUME_ALARM: 6
 ID_19241_13[...]22 > System.VOLUME_BLUETOOTH_SCO: 7
 ID_19241_13[...]22 > System.VOLUME_MUSIC: 11
 ID_19241_13[...]22 > System.VOLUME_NOTIFICATION: 5
 ID_19241_13[...]22 > System.VOLUME_RING: 5
 ID_19241_13[...]22 > System.VOLUME_SYSTEM: 7
 ID_19241_13[...]22 > System.VOLUME_VOICE: 4
 ID_19241_13[...]22 > System.WAIT_FOR_DEBUGGER: null
 ID_19241_13[...]22 > System.WALLPAPER_ACTIVITY: null
 ID_19241_13[...]22 > System.WIFI_SLEEP_POLICY: null
 ID_19241_13[...]22 > System.WIFI_STATIC_DNS1: null
 ID_19241_13[...]22 > System.WIFI_STATIC_DNS2: null
 ID_19241_13[...]22 > System.WIFI_STATIC_GATEWAY: null
 ID_19241_13[...]22 > System.WIFI_STATIC_IP: null
 ID_19241_13[...]22 > System.WIFI_STATIC_NETMASK: null
 ID_19241_13[...]22 > System.WIFI_USE_STATIC_IP: null
 ID_19241_13[...]22 > System.WINDOW_ANIMATION_SCALE: 1.0
 ID_19241_13[...]22 > SystemClock.elapsedRealtime(): 544213
 ID_19241_13[...]22 > SystemClock.uptimeMillis: 544214
 ID_19241_13[...]22 > GoogleServicesUtil.isGooglePlayServicesAvailable():FAILED
 ID_19241_13[...]22 > PackageManager.pm.getInstalledPackages() BEGIN_LIST:
 ID_19241_13[...]22 >   0: PackageInfo{41057618 android}
 ID_19241_13[...]22 >   1: PackageInfo{41057b00 com.android.backupconfirm}
 [...] (Enumeration of installed apps)
 ID_19241_13[...]22 >   56: PackageInfo{410349e0 jp.co.omronsoft.openwnn}
 ID_19241_13[...]22 > PackageManager.pm.getInstalledPackages() END_LIST:
 ID_19241_13[...]22 > CALLBACK_SignalStrength: 99 -1 -1 -1 -1 -1 -1 -1 -1 -1
-> 2147483647 -1 gsm|lte
 ID_19241_13[...]22 > DONE
```

Appendix C

Online Sandboxes for Android

List of all online sandboxes with the corresponding link.

Sandbox	**URL**	**Analyzed**
Andrubis	`http://anubis.iseclab.org/`	✓
ApkScan	`http://apkscan.nviso.be/`	✓
Foresafe	`http://www.foresafe.com/scan`	✓
JoeSandbox	`http://www.apk-analyzer.net/`	✓
Mobile-Sandbox (NG)	`http://mobilesandbox.org/`	✓
TraceDroid	`http://tracedroid.few.vu.nl/`	✓
AndroTotal	`http://andrototal.org/`	✗
App-Ray	`http://www.app-ray.de/`	✗
Aurasium	`http://www.aurasium.com/`	✗
CopperDroid	`http://copperdroid.isg.rhul.ac.uk/`	✗
Google Play Store	`https://play.google.com/`	✗
Sandroid	*Not available*	✗
VisualThreat	`http://www.visualthreat.com/`	✗
Yandex Store	`http://store.yandex.com/`	✗

Appendix D

Contents of the DVD

The DVD accompanying this thesis contains the following directories and data:

detection_mitigation contains the DroidBox patches to mitigate its detection.

droidbox contains the DroidBox images for different versions.

porting_patches contains the patches for DroidBox 4.4.

scripts contains developed scripts and small programs.

thesis contains the thesis as PDF document.

Bibliography

[1] Androidteam. Android system architecture diagram. Online, November 2009. https://code.google.com/p/androidteam/wiki/AndroidSystemArch, (Retrieved 11 January 2014).

[2] ARM Ltd. Arm - processor architecture. Online, 2014. http://www.arm.com/products/processors/instruction-set-architectures/index, (Retrieved 05 January 2014).

[3] ARM Ltd. Virtualization extensions. Online, 2014. http://www.arm.com/products/processors/technologies/virtualization-extensions.php, (Retrieved 23 Februray 2014).

[4] AV-Comparatives e.V. Cybercriminals infiltrate android markets. Technical report, AV-Comparatives e.V., 2013.

[5] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel. A view on current malware behaviors. In *USENIX workshop on large-scale exploits and emergent threats (LEET)*, 2009.

[6] Michael Becher, Felix C Freiling, Johannes Hoffmann, Thorsten Holz, Sebastian Uellenbeck, and Christopher Wolf. Mobile security catching up? revealing the nuts and bolts of the security of mobile devices. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 96–111. IEEE, 2011.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[8] Jean Bergeron, Mourad Debbabi, Jules Desharnais, Mourad M Erhioui, Yvan Lavoie, and Nadia Tawbi. Static detection of malicious code in executable programs. *Int. J. of Req. Eng*, 2001:184–189, 2001.

[9] Thomas Blasing, Leonid Batyuk, A-D Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An android application sandbox system for suspicious software detection. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 55–62. IEEE, 2010.

[10] Botnet Research Team, Xi'an Jiaotong University. Sanddroid. Online, 2013. http://sanddroid.xjtu.edu.cn/, (Retrieved 16 October 2013).

[11] Bundesverband des Deutschen Versandhandels. Umsatzzahlendes interaktiven handels im 3. quartal 2013. Online, November 2013. http://www.bvh.info/presse/pressemitteilungen/details/datum/2013/november/artikel/umsatzzahlen-des-interaktiven-handels-im-3-quartal-2013-steigerung-gegenueber-dem-3-quartal-des-v/?cHash=7f48b0fd0f885c14ae923ee211d14012, (Retrieved 03 February 2014).

[12] Daniel Bäumges. Intrusion detection of android applications by using taint analysis in an emulated environment. Master's thesis, Ruhr-Universität Bochum, 2012.

[13] Lorenzo Cavallaro, Prateek Saxena, and R Sekar. On the limits of information flow techniques for malware analysis and containment. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 143–163. Springer, 2008.

[14] Shuo Chen, Jun Xu, Nithin Nakka, Zbigniew Kalbarczyk, and Ravishankar K Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, pages 378–387. IEEE, 2005.

[15] Xu Chen, Jonathon Andersen, Zhuoqing Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177–186. IEEE, 2008.

[16] DroidBox Project Page. DroidBox, Android Application Sandbox. Online, August 2012. http://code.google.com/p/droidbox/, (Retrieved 03 September 2013).

[17] Benjamin Edelman. Secret ties in google's "open" android. Online, February 2014. http://www.benedelman.org/news/021314-1.html, (Retrieved 03 September 2013).

[18] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[19] eMarketer. Canada grabs greater share of ecommerce sales in north america. Online, September 2012. http://www.emarketer.com/Article/Canada-Grabs-Greater-Share-of-Ecommerce-Sales-North-America/1009328, (Retrieved 03 February 2014).

[20] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI*, volume 10, pages 255–270, 2010.

[21] Dave Evans. Top 25 technology predictions. Technical report, Cisco Internet Business Solution Group, 2009.

[22] F-Secure Corp. Mobile threat report q4 2011. Technical report, F-Secure Corp., 2011.

[23] F-Secure Corp. Mobile threat report q4 2012. Technical report, F-Secure Corp., 2012.

[24] F-Secure Corp. Mobile threat report q3 2013. Technical report, F-Secure Corp., 2013.

[25] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[26] Peter Ferrie. Attacks on more virtual machine emulators. Technical report, Symantec Corporation, 2007.

[27] Gartner Inc. Gartner says smartphone sales accounted for 55 percent of overall mobile phone sales in third quarter of 2013. Online, November 2013. https://www.gartner.com/newsroom/id/2623415, (Retrieved 22 February 2014).

[28] Laurent George, Valérie Viet Triem Tong, and Ludovic Mé. Blare tools: A policy-based intrusion detection system automatically set by the security policy. In *Recent Advances in Intrusion Detection*, pages 355–356. Springer, 2009.

[29] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland Schemers. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997.

[30] Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, and Ana Cavalli. Detecting control flow in smarphones: Combining static and dynamic analyses. In Yang Xiang, Javier Lopez, C.-C.Jay Kuo, and Wanlei Zhou, editors, *Cyberspace Safety and Security*, volume 7672 of *Lecture Notes in Computer Science*, pages 33–47. Springer Berlin Heidelberg, 2012.

[31] Sheran A. Gunasekera. *Android Apps Security*. Apress, 2012.

[32] Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Computer Security Applications Conference, 21st Annual*, pages 9–pp. IEEE, 2005.

[33] Christophe Hauser, Frédéric Tronel, Jason F Reid, and Colin J Fidge. A taint marking approach to confidentiality violation detection. In *Proceedings of the 10th Australasian Information Security Conference (AISC 2012)*, volume 125, pages 83–90. Australian Computer Society, 2012.

[34] Thorsten Holz, Markus Engelberth, and Felix Freiling. *Learning More About the Underground Economy: a Case-study of Keyloggers and Dropzones*. Springer, 2009.

[35] Honeynet, DroidBox. DroidBox: alpha release. Online, July 2011. http://honeynet.org/node/744, (Retrieved 03 September 2013).

[36] International Telecommunications Union. Key 2006-2013 ict data for the world. Online, 2013. http://www.itu.int/en/ITU-D/Statistics/Documents/statistics/2013/ ITU_Key_2005-2013_ICT_data.xls, (Retrieved 31 January 2014).

[37] Ipsos Public Affairs. 2010 MAAWG Email Security Awareness and Usage Report. Technical report, Messaging Anti-Abuse Working Group, march 2010.

[38] Kaspersky Lab. Kaspersky security bulletin 2013. Technical report, Kaspersky Lab, 2013.

[39] Paul L Kirk. Crime investigation; physical evidence and the police laboratory. 1953.

[40] Paul C Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and other Systems. In *Advances in Cryptology—CRYPTO'96*, pages 104–113. Springer, 1996.

[41] Neelie Kroes. Internet security: Everyone's responsibility. Speech, 48th Munich Security Conference, February 2012. http://europa.eu/rapid/press-release_SPEECH-12-68_en.htm, (Retrieved 28 January 2014).

[42] Patrik Lantz. An android application sandbox for dynamic analysis. Master's thesis, Lund University, 2011.

[43] Steffen Liebergeld and Matthias Lange. Android security, pitfalls and lessons learned. In *Information Sciences and Systems 2013*, pages 409–417. Springer, 2013.

[44] Hiroshi Lockheimer. Android and security. Online, February 2012. http://googlemobile.blogspot.de/2012/02/android-and-security.html?m=1, (Retrieved 26 February 2014).

[45] Mark Lutz. *Programming Python*. O'Reilly Media, 2010.

[46] Federico Maggi, Andrea Valdi, and Stefano Zanero. Andrototal: a flexible, scalable toolbox and service for testing mobile malware detectors. In *Proceedings of the Third ACM workshop on Security and privacy in smartphones & mobile devices*, pages 49–54. ACM, 2013.

[47] Denis Maslennikov. Mobile malware evolution: An overview, part 4. Online, 2011. http://www.securelist.com/en/analysis/204792168/Mobile_Malware_Evolution_An_Overview_Part_4, (Retrieved 05 January 2014).

[48] F. Matenaar and P. Schulz. Detecting Android Sandboxes. Online, August 2012. http://www.dexlabs.org/blog/btdetect, (Retrieved 05 September 2013).

[49] McAfee Inc. Consumer mobile trends report - june 2013. Technical report, McAfee Inc., June 2013.

[50] McAfee Inc. Mcafee labs threats report: Third quater 2013. Technical report, McAfee Inc., 2013.

[51] Mobile Sandbox. Mobile sandbox. Online. http://mobilesandbox.org/, (Retrieved 23 January 2014).

[52] Andreas Moser, Christopher Kruegel, and Engin Kirda. Exploring multiple execution paths for malware analysis. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 231–245. IEEE, 2007.

[53] Andreas Moser, Christopher Kruegel, and Engin Kirda. Limits of static analysis for malware detection. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, pages 421–430. IEEE, 2007.

[54] Srijith K Nair, Patrick ND Simpson, Bruno Crispo, and Andrew S Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197(1):3–16, 2008.

[55] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[56] J Oberheide and C Miller. Dissecting the android bouncer. *SummerCon2012, New York*, 2012.

[57] Ofcome. International communications market report. Technical report, Ofcome, 2013.

[58] Open Handset Alliance. Android 4.2 apis. Online, 2014. developer.android.com/about/versions/android-4.2.html, (Retrieved 05 March 2014).

[59] Open Handset Alliance. Android 4.3 apis. Online, 2014. developer.android.com/about/versions/android-4.3.html, (Retrieved 05 March 2014).

[60] Open Handset Alliance. Android 4.4 apis. Online, 2014. developer.android.com/about/versions/android-4.4.html, (Retrieved 05 March 2014).

[61] Open Handset Alliance. Dalvik technical information. Online, 2014. http://source.android.com/devices/tech/dalvik/index.html, (Retrieved 23 January 2014).

[62] Open Handset Alliance. Sdk release notes. Online, 2014. https://developer.android.com/sdk/RELEASENOTES.html, (Retrieved 23 January 2014).

[63] Open Handset Alliance. Platform versions. Online, January 2014. https://developer.android.com/about/dashboards/index.html, (Retrieved 28 January 2014).

[64] Oracle Corporation. Mobility solutions for oracle applications. Online, 2013. http://www.oracle.com/us/products/applications/036048.htm, (Retrieved 18 February 2014).

[65] Percoco and Schulte. Adventures in bouncerland: Failures of automated malware detection within mobile application markets. *Black Hat USA 2012*, 2012.

[66] Enrico Perla and Massimiliano Oldani. *A Guide to Kernel Exploitation: Attacking the Core*. Elsevier, 2010.

[67] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Commun. ACM*, 17(7):412–421, July 1974.

[68] Stefan Porteck. State of the art. *c't - Magazin für Computertechnik*, 05:144f, February 2014.

[69] Thomas Raffetseder, Christopher Krügel, and Engin Kirda. Detecting system emulators. In *Information Security*, pages 1–18. Springer, 2007.

[70] John S Robin and Cynthia E Irvine. Analysis of the intel pentium's ability to support a secure virtual machine monitor. Technical report, DTIC Document, 2000.

[71] SAP AG. Enterprise mobility: Mobile applications, platforms, and services. Online, 2013. http://www.sap.com/pc/tech/mobile.html, (Retrieved 18 February 2014).

[72] 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, 2013.

[73] Patrick Schulz. Android emulator detection by observing low-level caching behavior. Online, December 2013.

[74] Secunia. Secunia vulnerability review. Technical report, Secunia, May 2013.

[75] M. Sikorski and A. Honig. *Practical Maleware Analysis*. No Starch Press, 2012.

[76] Jeff Six. *Application Security for the Android Platform*. O'Reilly Media, 2011.

[77] Robin Sommer and Vern Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 305–316. IEEE, 2010.

[78] Dawn Song, David Brumley, Heng Yin, Juan Caballero, Ivan Jager, Min Gyung Kang, Zhenkai Liang, James Newsome, Pongsin Poosankam, and Prateek Saxena. Bitblaze: A new approach to computer security via binary analysis. In *Information systems security*, pages 1–25. Springer, 2008.

[79] Michael Spreitzenbarth. *Dissecting the Droid: Forensic Analysis of Android and its malicious Applications*. PhD thesis, Friedrich-Alexander-Universität, 2013.

[80] Michael Spreitzenbarth and Felix Freiling. Android malware on the rise. Technical Report CS-2012,4, Technische Fakultät -ohne weitere Spezifikation-, 2012.

[81] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1808–1815, New York, NY, USA, 2013. ACM.

[82] Statista. E-Commerce-Umsatz mit Waren in Deutschland in den Jahren von 2000 bis 2012 und Prognose für 2013 (in Milliarden Euro). Online, November 2013. http://de.statista.com/statistik/daten/studie/71568/umfrage/online-umsatz-mit-waren-seit-2000/, (Retrieved 17 February 2014).

[83] Statista. Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 3rd quarter 2013. Online, November 2013. http://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/, (Retrieved 22 February 2014).

[84] T. Strazzere. Detecting an emulator, and getting around detection. Online, September 2010. http://www.strazzere.com/blog/2010/09/detecting-an-emulator-and-getting-around-them-it/, (Retrieved 05 September 2013).

[85] Symantec Corporation. W32.Mydoom.M@mm. Online, July 2004. http://www.symantec.com/securityresponse/writeup.jsp?docid=2004-072615-3527-99, (Retrieved 02 December 2013).

[86] Symantec Corporation. Internet security threat report 2013. Technical report, Symantec Corporation, 2013.

[87] The World Bank. Internet users (per 100 people). Online, 2013. http://data.worldbank.org/indicator/IT.NET.USER.P2, (Retrieved 31 January 2014).

[88] David Thomas, Chad Fowler, and Andrew Hunt. *Programming Ruby*, volume 13. Pragmatic Bookshelf, 2004.

[89] Rob Thomas and Jerry Martin. The underground economy: Priceless. *USENIX; login*, 31(6):7–16, 2006.

[90] Philipp Trinius, Thorsten Holz, Jan Gobel, and Felix C Freiling. Visual analysis of malware behavior using treemaps and thread graphs. In *Visualization for Cyber Security, 2009. VizSec 2009. 6th International Workshop on*, pages 33–38. IEEE, 2009.

[91] Twitter Inc. Twitter. Online, 2014. http://twitter.com/, (Retrieved 1 April 2014).

[92] Victor van der Veen, Herbert Bos, and Christian Rossow. Dynamic analysis of android malware. Master's thesis, University Amsterdam, 2013.

[93] Vienna University of Technology. Andrubis - Analysis of Android APKs. Online. http://anubis.iseclab.org, (Retrieved 17 December 2013).

[94] VMware, Inc. Vmware. Online, 2014. http://www.vmware.com/, (Retrieved 15 January 2014).

[95] Larry Wall, Tom Christiansen, and Jon Orwant. *Programming Perl*. O'Reilly Media, 2000.

[96] Christina Warren. Google play hits 1 million apps. Online, July 2013. http://mashable.com/2013/07/24/google-play-1-million/, (Retrieved 28 January 2014).

[97] WhatsApp Inc. Whatsapp. Online, 2014. http://www.whatsapp.com/, (Retrieved 2 March 2014).

[98] Carsten Willems and Felix C Freiling. Reverse code engineering-state of the art and countermeasures. *it-Information Technology*, 54(2):53–63, 2012.

[99] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *Security & Privacy, IEEE*, 5(2):32–39, 2007.

[100] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

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

[102] J. Zhuge et al. Studying malicious websites and the underground economy on the chinese web. Technical report, Peking University, University of Mannheim, 2007.