VU University Amsterdam
Faculty of Sciences
Department of Computer Sciences

Internet & Web Technology
Master thesis

Dynamic Analysis of Android Malware

Victor van der Veen

supervisors
prof. dr. ir. Herbert Bos
dr. Christian Rossow

August 31, 2013

Abstract

Expecting a shipment of 1 billion Android devices in 2017, cyber criminals have naturally extended their vicious activities towards Google's mobile operating system: threat researchers are reporting an alarming increase of detected Android malware from 2012 to 2013. In order to have some control over the estimated 700 new Android applications that are being released every day, there is need for a form of automated analysis to quickly detect and isolate new malware instances.

We present the TraceDroid Analysis Platform, a scalable, automated framework for dynamic analysis of Android applications to detect suspicious, possibly malicious apps using a comprehensive method tracing scheme dubbed TraceDroid. We provide means to aid further post-analysis on suspects to allow malware researchers to fully understand their behavior and ultimately label them as malicious or benign. Our framework can therefore aid and direct scarce analysis resources towards applications that have the greatest potential of being malicious.

We show that TraceDroid is almost 50% faster than Android's original profiler implementation while revealing much more detail about the app's execution. This makes it a perfect tool not only for malware analysts, but also for app developers and reverse engineers. For a random set of 35 both benign and malicious samples, the stimulation engine of our TraceDroid Analysis Platform achieves an average code coverage of 33.6% which is even more than when they are stimulated manually (32.9%).

Contents

1 Introduction **7**

2 Background Information **9**
2.1 Android System Architecture 9
2.1.1 Linux kernel 9
2.1.2 Libraries 10
2.1.3 Android runtime 10
2.1.4 Application framework 11
2.1.5 Applications 12
2.2 Dalvik Virtual Machine 12
2.2.1 Hardware constraints 12
2.2.2 Bytecode 13
2.3 Apps 13
2.3.1 Application components 14
2.3.2 Manifest 15
2.3.3 Native code 15
2.3.4 Distribution 16
2.4 Malware 16
2.4.1 Types of malware 16
2.4.2 Malware distribution 17
2.4.3 Malware data sets 18

3 Design **19**
3.1 Design 19
3.1.1 What to collect 19
3.1.2 Framework design 20
3.1.3 Native code 21
3.2 Specification 21
3.2.1 Specification 21
3.2.2 Existing solutions 23

4 Implementation **24**
4.1 TraceDroid 24
4.1.1 Implementation 25
4.1.2 Andrubis integration 30
4.1.3 Discussion 32
4.2 Android Framework Modifications 32
4.2.1 `killProcess()` 32

4.2.2 Making `am profile stop` blocking 32
4.2.3 Timeout values 35
4.3 Analysis Framework 35
4.3.1 Static analysis 35
4.3.2 Dynamic analysis 35
4.3.3 Post processing 36
4.3.4 Inspecting TraceDroid output 38
4.4 Bytecode Weaving 39
4.4.1 AOP: Aspect Oriented Programming 40
4.4.2 Advantages and drawbacks of bytecode weaving 42

5 Evaluation 45
5.1 Benchmarking TraceDroid 45
5.1.1 Benchmark setup 45
5.1.2 Benchmark results 46
5.2 Benchmarking TraceDroid + Andrubis 48
5.2.1 Andrubis background 48
5.2.2 Benchmark results 49
5.3 Coverage 51
5.3.1 Compared to manual analysis 51
5.3.2 Breakdown of simulation actions 54
5.3.3 Coverages results 55
5.4 Failures 58
5.5 Dissecting Malware 60
5.5.1 ZitMo: ZeuS in the Mobile 61
5.5.2 Dissecting `a1593777ac80b828d2d520d24809829d` 61
5.5.3 Discussion 66

6 Related Work 68
6.1 Background and Surveys 68
6.2 Systematization of Knowledge 69
6.2.1 Attributes 69
6.2.2 Classification 71
6.2.3 Overview of (proposed) frameworks 72
6.3 Dynamic Analysis Platforms 78
6.3.1 AASandbox 78
6.3.2 TaintDroid 78
6.3.3 DroidBox 78
6.3.4 Bouncer 79
6.3.5 Andrubis 80
6.3.6 DroidScope 80
6.3.7 AppsPlayground 80
6.3.8 Mobile-Sandbox 80
6.3.9 CopperDroid 81
6.3.10 Closed frameworks 81

7 Conclusions **82**
7.1 Future Work 82
7.1.1 TraceDroid 82
7.1.2 TraceDroid Analysis Platform 85
7.1.3 Other research directions 86
7.2 Conclusions 87
7.2.1 TraceDroid 87
7.2.2 TraceDroid Analysis Platform 87

Appendices **96**

A Sample Set **97**

B Availability of Related Work **106**

List of Figures

2.1 Android low level system architecture 10
2.2 ANR dialog 11
2.3 Android application build process 13

3.1 Design for Android dynamic analysis platform 20

4.1 Example stack layout 27
4.2 Android source code control flow diagram for disabling method tracing 34
4.3 Weave process 42

5.1 Benchmark results 47
5.2 CDF for TraceDroid coverage results 56
5.3 Code coverage breakdown per simulation 57
5.4 ZitMo 62
5.5 Call graph for ZitMo 67

List of Tables

4.1 Possible race condition in LOGD_TRACE 28
4.2 Description of default actions simulated during analysis 36
4.3 Excluded libraries for naive code coverage computation 37
4.4 Description of different feature sets extracted 38
4.5 Common fields for Function and Constructor objects 38
4.6 Variables for direct access 39
4.7 Options for generate_callgraph() 39

5.1 Benchmark results (all times in ms) 47
5.2 Overview of operations detected by ANDRUBIS 48
5.3 ANDRUBIS similarities for different runtime values (without repetition) 49
5.4 ANDRUBIS coverage results for different runtime values 50
5.5 ANDRUBIS similarities for different runtime values (equal data field required) 50
5.6 Coverage results for benign and malicious samples 52
5.7 ANDRUBIS breakdown 54
5.8 TRACEDROID breakdown 55
5.9 Code coverage results 56
5.10 Classification of detected failures 58

6.1 Overview of (proposed) frameworks 73

A.1 Benign sample set 97
A.2 Malicious sample set 101

B.1 Availability of research frameworks 106

List of Listings

3.1 Source code for a very simple Android app 22
3.2 Desired trace output 23
4.1 Method trace for thrown exceptions 29
4.2 Actual trace output 31
4.3 Enabling method tracing using AOP 40
4.4 Minimal method tracing aspect 41
4.5 Trace aspect output 43
5.1 Stack trace with added method resolution for the **unknown** bug 60
5.2 Generating a feature set for ZitMo 62
5.3 `getResponseCode()` invocation 63
5.4 Retrieving URL parameters 63
5.5 Domain name deobfuscation 63
5.6 Method trace for `GetLastSms()` 64
5.7 Method trace for `AlternativeControl()` 64
5.8 Manual dynamic analysis 65
5.9 Method trace for `AlternativeControl()` 65
7.1 Android application using reflection 83

Chapter 1

Introduction

With an estimated market share of 70% to 80%, Android has become the most popular operating system for smartphones and tablets [12, 43]. Expecting a shipment of 1 billion Android devices in 2017 and with over 50 billion total app downloads since the first Android phone was released in 2008, cyber criminals naturally expanded their vicious activities towards Google's mobile platform. Mobile threat researchers indeed recognize an alarming increase of Android malware from 2012 to 2013 and estimate that the number of detected malicious apps is now in the range of 120,000 to 718,000 [1, 30, 38, 65]. In the summer of 2012, the sophisticated *Eurograbber* attack showed that mobile malware may be a very lucrative business by stealing an estimated €36,000,000 from bank customers in Italy, Germany, Spain and the Netherlands [39].

Android's open design allows users to install applications that do not necessarily originate from the Google Play Store. With over 1 million apps available for download via Google's official channel [68], and possibly another million spread among third-party app stores, we can estimate that there are over 20,000 new applications being released every month. This requires malware researchers and app store administrators to have access to a scalable solution for quickly analyzing new apps and identifying and isolating malicious applications.

Google reacted to the growing interest of miscreants in Android by revealing BOUNCER in February 2012, a service that checks apps submitted to the Google Play Store for malware [44]. However, research has shown that BOUNCER's detection rate is still fairly low and that it can easily be bypassed [37, 48]. A large body of similar research on Android malware has been proposed, but none of them provide a comprehensive solution to obtain a thorough understanding of unknown applications: Bläsing et al. and Reina et al. limit their research to system call analysis [7, 58], Enck et al. focuses on taint tracking [26], Rastogi et al. and Spreitzenbarth et al. track only specific API invocations [56, 64], and work done by Yan and Yin is bound to use an emulator [72].

In this work, we present a scalable *dynamic* analysis platform for Android applications to detect suspicious, possibly malicious applications. We provide means to aid further post-analysis on these suspects to allow malware researchers to fully understand their behavior. By using dynamic analysis, we have the advantage that our results are not hindered by obfuscation techniques used by the application, unlike static analysis approaches. Running the application in a sandboxed environment allows us to keep track of an app's entire control flow

without having to apply complex decompilation and deobfuscation techniques.

We introduce a modified Android OS dubbed TraceDroid to generate comprehensive method traces for a given Android application. In addition, we present the TraceDroid Analysis Platform (TAP) that automatically executes and stimulates unknown applications within TraceDroid. We provide a number of plug-ins for TAP that perform post-analysis on TraceDroid's output, including generating a fingerprint of an app's execution trace as well as computing the amount of code covered during dynamic stimulation. Moreover, results of these plug-ins may be used by a machine learning algorithm to classify and detect malware or to evaluate the effectiveness of TAP. Finally, TraceDroid has been integrated into Andrubis, a popular online platform for analysis of Android applications. We analyzed numerous of both benign and malicious applications to ensure TraceDroid and TAP do not interfere an app's normal execution behavior.

To summarize, we present the following contributions.

- We present TraceDroid, a modified version of Android's Dalvik Virtual Machine that provides comprehensive method trace output. We show that TraceDroid outperforms Android's existing method tracer in terms of performance, while revealing great detail on an app's behavior, including invoked Java methods with parameter resolution and return values as well as textual representations of objects used during the app's runtime.
- We introduce the TraceDroid Analysis Platform (TAP), a framework that uses TraceDroid to perform dynamic analysis of unknown Android applications. TAP aims to maximize the observed malware behavior by simulating certain events and includes a number of plug-ins to ease post-analysis of unknown applications, as well as to measure the effectiveness of the executed dynamic analysis.
- We provide a detailed overview of existing work on the field of Android security: using a number of characteristics, we classify research efforts into 7 categories.

This document is further outlined as follows. In Chapter 2, we provide an introduction into the Android architecture and outline the techniques used by mobile malware authors. In Chapter 3, we define the scope of our work and provide a specification of our TraceDroid implementation combined with desired output. The implementation notes of TraceDroid and TAP are discussed in Chapter 4. We evaluate both implementations in Chapter 5. In Chapter 6, we discuss related research efforts. We look closely at related work that uses dynamic analysis, but also outline a systematization of knowledge wherein we classify known Android security research efforts. Finally, in Chapter 7, we propose a number of future research directions and possible extensions to our implementations and conclude our work.

Chapter 2

Background Information

Before we discuss the details of our analysis framework, it is important to understand how Android and Android applications work. In this chapter, we provide a short introduction into the Android architecture.

We start with a high level overview of the Android system architecture in Section 2.1. In this section, we describe the implementation design of Android and discuss its various component layers.

Since a major part of our contribution focuses on modifying the virtual machine that is responsible for executing Android applications, we discuss this layer in more detail in Section 2.2.

An overview of the core components found in Android applications is outlined in Section 2.3. This section discusses *activities*, *services*, *receivers*, and *intents*, the building blocks of Android applications.

Finally, in Section 2.4, we briefly discuss how Android malware takes advantage of the Android platform.

2.1 Android System Architecture

The Android software stack is illustrated in Figure 2.1[1]. In this figure, green items are components written in native code (C/C++), while blue items are Java components interpreted and executed by the *Dalvik Virtual Machine*. The bottom red layer represents the Linux kernel components and runs in kernel space.

In the following subsections, we briefly discuss the various abstraction layers using a bottom-up approach. For a more detailed overview, we refer to existing studies [9, 22].

2.1.1 Linux kernel

Android uses a specialized version of the *Linux Kernel* with a few special additions. These include wakelocks (mechanisms to indicate that apps need to have the device stay on), a memory management system that is more aggressive in preserving memory, the Binder IPC driver, and other features that are important for a mobile embedded platform like Android.

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

Figure 2.1: Android low level system architecture

2.1.2 Libraries

A set of native C/C++ libraries is exposed to the *Application Framework* and *Android Runtime* via the *Libraries* component. These are mostly external libraries with only very minor modifications such as OpenSSL[2], WebKit[3] and bzip2[4]. The essential C libraries, codename Bionic, were ported from BSD's libc and were rewritten to support ARM hardware and Android's own implementation of pthreads based on Linux futexes.

2.1.3 Android runtime

The middleware component called *Android Runtime* consists of the *Dalvik Virtual Machine* (Dalvik VM or DVM) and a set of *Core Libraries*. The Dalvik VM is responsible for the execution of applications that are written in the Java programming language and is discussed in more detail in Section 2.2. The core libraries are an implementation of general purpose APIs and can be used by the applications executed by the Dalvik VM. Android distinguishes two categories of core libraries.

- Dalvik VM-specific libraries.
- Java programming language interoperability libraries.

[2] `http://www.openssl.org`
[3] `http://www.webkit.org`
[4] `http://www.bzip.org`

The first set allow in processing or modifying VM-specific information and is mainly used when bytecode needs to be loaded into memory. The second category provides the familiar environment for Java programmers and comes from Apache's Harmony[5]. It implements most of the popular Java packages such as `java.lang` and `java.util`.

2.1.4 Application framework

The *Application Framework* provides high level building blocks to applications in the form of various `android.*` packages. Most components in this layer are implemented as applications and run as background processes on the device. Some components are responsible for managing basic phone functions like receiving phone calls or text messages or monitoring power usage. A couple of components deserve a bit more attention:

Activity Manager The *Activity Manager* (AM) is a process-like manager that keeps track of active applications. It is responsible for killing background processes if the device is running out of memory. It also has the capability to detect unresponsive applications when an app does not respond to an input event within 5 seconds (such as a key press or screen touch). It then prompts an *Application Not Responding* (ANR) dialog (shown in Figure 2.2).

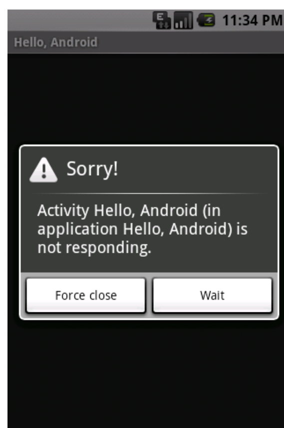


Figure 2.2: ANR dialog

Content Providers *Content Providers* are one of the primary building blocks for Android applications. They are used to share data between multiple applications. Contact list data, for example, can be accessed by multiple applications and must thus be stored in a content provider.

Telephony Manager The *Telephony Manager* provides access to information about the telephony services on the device such as the phone's unique device identifier (IMEI) or the current cell location. It is also responsible for managing phone calls.

Location Manager The *Location Manager* provides access to the system location services which allow applications to obtain periodic updates of the device's geographical location by using the device's GPS sensor.

[5] `http://harmony.apache.org`

2.1.5 Applications

Applications or *apps* are built on top of the *Application Framework* and are responsible for the interaction between end-users and the device. It is unlikely that an average user ever has to deal with components not in this layer. Pre-installed applications offer a number of basic tasks a user would like to perform (making phone calls, browsing the web, reading e-mail, etc.), but users are free to install third-party applications to use other features (e.g., play games, watch videos, read news, use GPS navigation, etc.). We discuss Android applications in more detail in Section 2.3.

2.2 Dalvik Virtual Machine

Android's design encourages software developers to write applications that offer users extra functionality. Google decided to use Java as the platform's main programming language as it is one of the most popular languages: Java has been the number one programming language almost continuously over the last decade[6], and a large number of development tools are available for it (e.g., Eclipse[7] and NetBeans[8]). Java source code is normally compiled to and distributed as Java bytecode which, at runtime, is interpreted and executed by a Virtual Machine (VM). For Android, however, Google decided to use a different bytecode and VM format named Dalvik. During the compilation process of Android applications, Java bytecode is converted to Dalvik bytecode which can later be executed by the specially designed Dalvik VM.

Since a large part of our contributions involve modifying the Dalvik VM, we now discuss it in a bit more detail.

2.2.1 Hardware constraints

The Android platform was specifically designed to run on mobile devices and thus comes has to overcome some challenging hardware restrictions when compared to regular desktop operating systems: mobile phones are limited in size and are powered by only a battery. Due to this mobile character, initial devices contained a relatively slow CPU and had only little amount of RAM left once the system was booted. Despite these ancient specifications, the Android platform does rely on modern OS principles: each application is supposed to run in its own process and has its own memory space which means that each application should run in its own VM.

It was argued that the hardware constraints, made it hard to fulfill the security requirements using existing Java virtual machines [8]. To overcome these issues, Android uses the Dalvik VM. A special instance of the DVM is started at boot time which will become the parent of all future VMs. This VM is called the *Zygote* process and preloads and preinitializes all system classes (the *core libraries* discussed in Section 2.1.3). Once started, it listens on a socket and `fork()`s on command whenever a new application start is requested. Using `fork()` instead of starting a new VM from scratch increases the speedup time

[6]`http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`
[7]`http://www.eclipse.org`
[8]`http://www.netbeans.org`

and by sharing the memory pages that contain the preloaded system classes, Android also reduces the memory footprint for running applications.

Furthermore, as opposed to regular stack-based virtual machines — a mechanism that can be ported to any platform — the DVM is register-based and is designed to specifically run on ARM processors. This allowed the VM developers to add more speed optimizations.

2.2.2 Bytecode

The bytecode interpreted by the DVM is so-called DEX bytecode (Dalvik EXecutable code). DEX code is obtained by converting Java bytecode using the `dx` tool. The main difference between the DEX file format and Java bytecode is that all code is repacked into one output file (`classes.dex`), while removing duplicate function signatures, string values and code blocks. Naturally, this results in the use of more pointers within DEX bytecode than in Java `.class` files. In general, however, `.dex` files are about 5% smaller than their counterpart, compressed `.jar` files.

It is worth mentioning that during the installation of an Android application, the included `classes.dex` file is verified and optimized by the OS. Verification is done to reduce runtime bugs and to make sure that the program cannot misbehave. Optimization involves static linking, inlining of special (native) methods (e.g. calls to `equals()`), and pruning empty methods.

2.3 Apps

Android applications are distributed as Android Package (APK) files. APK files are signed ZIP files that contain the app's bytecode along with all its data, resources, third-party libraries and a manifest file that describes the app's capabilities. Figure 2.3 shows the simplified process of how Java source code projects are translated to APK files.

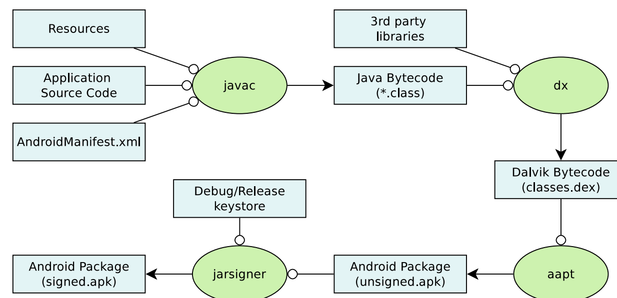


Figure 2.3: Android application build process

To improve security, apps run in a sandboxed environment. During installation, applications receive a unique Linux user ID from the Android OS. Permissions for files in an application are then set so that only the application itself has access to them. Additionally, when started, each application is granted its own VM which means that code is isolated from other applications. It is stated by the Android documentation that this way, Android implements the *principle of*

least privilege as each application has access to only the components it requires to do its work[9].

2.3.1 Application components

We now outline a number of core application components that are used to build Android apps. For more information on Android application fundamentals, we refer to the official documentation[10].

Activities

An *activity* represents a single screen with a particular user interface. Apps are likely to have a number of activities, each with a different purpose. A music player, for instance, might have one activity that shows a list of available albums and another activity to show the song that is currently be played with buttons to pause, enable shuffle, or fast forward. Each activity is independent of the others and, if allowed by the app, can be started by other applications. An e-mail client, for example, might have the possibility to start the music app's play activity to start playback of a received audio file.

Services

Services are components that run in the background to perform long-running operations and do not provide a user interface. The music application, for example, will have a music service that is responsible for playing music in the background while the user is in a different application. Services can be started by other components of the app such as an activity or a broadcast receiver.

Content providers

Content providers are used to share data between multiple applications. They manage a shared set of application data. Contact information, for example, is stored in a content provider so that other applications can query it when necessary. A music player may use a content provider to store information about the current song being played, which could then be used by a social media app to update a user's 'current listening' status.

Broadcast receivers

A *broadcast receiver* listens for specific system-wide broadcast announcements and has the possibility to react upon these. Most broadcasts are initiated from the system and announce that, for example, the system completed the boot procedure, the battery is low, or an incoming SMS text message was received. Broadcast receivers do not have a user interface and are generally used to act as a gateway to other components. They might, for example, initiate a background service to perform some work based on a specific event.

Two types of broadcasts are distinguished: non-ordered and ordered. Non-ordered broadcast are sent to all interested receivers at the same time. This means that a receiver cannot interfere with other receivers. An example of such

[9] `http://developer.android.com`
[10] `http://developer.android.com/guide/components/fundamentals.html`

broadcast is the battery low announcement. Ordered broadcasts, on the other hand, are first passed to the receiver with the highest priority, before being forwarded to the receiver with the second highest priority, etc. An example for this is the incoming SMS text message announcement.

Broadcast receivers that receive ordered broadcasts can, when done processing the announcement, decide to abort the broadcast so that it is not forwarded to other receivers. In the example of incoming text messages, this allows vendors to develop an alternative text message manager that can disable the existing messaging application by simply using a higher priority receiver and aborting the broadcast once it finished handling the incoming message.

Intents

Activities, services and broadcast receivers are activated by an asynchronous message called an intent. For activities and services, intents define an action that should be performed (e.g., `view` or `send`). They may include additional data that specifies what to act on. A music player application, for example, may send a `view` intent to a browser component to open a webpage with information on the currently selected artist.

For broadcast receivers, the intent simply defines the current announcement that is being broadcast. For an incoming SMS text message, the additional data field will contain the content of the message and the sender's phone number.

2.3.2 Manifest

Each Android application comes with an `AndroidManifest.xml` file that informs the system about the app's components. Activities and services that are not declared in the manifest can never run. Broadcast receivers, however, can be either declared in the manifest or may be registered dynamically via the `registerReceiver()` method. The manifest also specifies application requirements such as special hardware requirements (e.g., having a camera or GPS sensor), or the minimal API version necessary to run this app.

In order to access protected components (e.g., camera access, or access to the user's contact list), an application needs to be granted permission. All necessary permissions must be defined in the app's `AndroidManifest.xml`. This way, during installation, the Android OS can prompt the user with an overview of used permissions after which a user explicitly has to grant the app access to use these components.

Within the OS, protected components are element of a unique Linux group ID. By granting an app permissions, it's VM becomes a member of the accompanying groups and can thus access the restricted components.

2.3.3 Native code

It may be helpful for certain types of applications to use native code languages like C and C++ so that they can reuse existing code libraries written in these languages. Typical good candidates for native code usage are self-contained, CPU intensive operations such as signal processing, game engines, and so on. Unlike Java bytecode, native code runs directly on the processor and is thus not interpreted by the Dalvik VM.

2.3.4 Distribution

Android users are free to install any (third-party) application via the Google Play Store (previously known as the Android Market). Google Play is an online application distribution platform where users can download and install free or paid applications from various developers (including Google self). To protect the Play Store from malicious applications, Google uses an in-house developed automated anti-virus system named Google Bouncer (discussed in more detail in Chapter 6).

Users have the possibility to install applications from other sources than Google Play. For this, a user must enable the *unknown sources* option in the device's settings overview and explicitly accepts the risks of doing so. By using external installation sources, users can install APK files downloaded from the web directly, or choose to use third-party markets. These third-party markets sometimes offer a specialized type of applications, such as MiKandi's Adult app store[11], or target users from specific countries, like Chinese app stores Anzhi[12] and Xiaomi[13] (a popular Chinese phone manufacturer).

2.4 Malware

Recent reports focusing on mobile malware trends estimate that the number of malicious Android apps is now in the range of 120,000 to 718,000 [1, 30, 38, 65]. In this section, we take a closer look at mobile malware characteristics, how they are distributed and what data sets are publicly available for malware researchers.

2.4.1 Types of malware

The majority of Android malware can be categorized in two types, both using social engineering to trick users into installing the malicious software.

Fake install/SMS trojan The majority of Android malware is classified as *fake installers* or *SMS trojans*. These apps pretend to be an installer for legitimate software and trick users into installing them on their devices. When executed, the app may display a service agreement and, once the user has agreed, sends premium rated text messages. The promised functionality is almost never available. Variants include repackaged applications that provide the same functionality as the original — often paid — app, but have additional code to secretly send SMS messages in the background .

SMS trojans are relatively easy to implement: only a single main activity with a button that initiates the sending of an SMS message when clicked is required. It is estimated that on average, each deployed sample generates an immediate profit of around $10 USD [38]. This type of attack is also referred to as *toll fraud*. High profit and easy manufacturing make toll fraud apps popular among malware authors.

[11]`http://www.mikandi.com`
[12]`http://www.anzhi.com`
[13]`http://app.xiaomi.com`

Spyware/Botnet Another observed type of Android malware is classified as *spyware* and has capabilities to forward private data to a remote server. In a more complex form, the malware could also receive commands from the server to start specific activities in which case it is part of a *botnet*. Spyware is likely to use some of the components described in Section 2.3.1. *Broadcast receivers* are of particular interest as they can be used to secretly intercept and forward incoming SMS messages to a remote server or to wait for `BOOT_COMPLETED` to start a background service as soon as the device is started.

In the summer of 2012, the sophisticated *Eurograbber* attack showed that these type of malware may be very lucrative by stealing an estimated €36,000,000 from bank customers in Italy, Germany, Spain and the Netherlands [39].

2.4.2 Malware distribution

A problem with third-party marketplaces described in Section 2.3.4, is the lack of accountability. There are often no entry limitations for mobile app developers which results in poor and unreliable applications being pushed to these stores and making it to Android devices. Juniper Networks finds that malicious applications often originate from these marketplaces, with China (173 stores hosting some malware) and Russia (132 'infected' stores) being the world's leading suppliers [38].

One of the issues Android has to deal with in respect to malware distribution is the loose management of the devices. Over the past few years, Android versions have become fragmented, with only 6.5% of all devices running the latest Android version 4.2 (codename Jelly Bean). More than two years after its first release in February 2011, a majority of Android devices (33.0%) is still running Android 2.3.3–2.3.7 (codename Gingerbread)[14]. This fragmentation makes new security features only available to a small group of users who happen to use the latest Android release. Any technique invented to prevent malicious behavior will never reach the majority of Android users, until they buy a new device.

One of the security enhancements in Android 4.2, for example, is the **more control of premium SMS** feature[15]. This feature notifies the user when an application tries to send an SMS message that might cause additional charges. This feature would prevent a large portion of the previously discussed SMS trojans, but is unfortunately not attainable for the majority of Android users.

New Android releases also come with bugfixes for core components to prevent against arbitrary code execution exploits. Android versions prior to 2.3.7 are especially vulnerable to these root exploits (examples include *rage against the cage*[16], *exploid*[17] and *zergRush*[18]). While these exploits were originally developed to overcome limitations that carriers and hardware manufactures put on some devices, they have also been used by malware to obtain a higher privilege level without a user's consent. This approach allows malware to request only a few permissions during app installation, but still access the entire system once the app is started.

[14] `http://developer.android.com/about/dashboards/index.html`
[15] `http://source.android.com/devices/tech/security/enhancements.html`
[16] `http://dtors.org/2010/08/25/reversing-latest-exploid-release`
[17] `http://thesnkchrmr.wordpress.com/2011/03/27/udev-exploit-exploid`
[18] `http://github.com/revolutionary/zergRush`

2.4.3 Malware data sets

Public access to known Android malware samples is mainly provided via the Android Malware Genome Project[19] and Contagio Mobile[20]. The malgenome-project was a result of the work done by Zhou and Jiang [78] and contains over 1200 Android malware samples, classified in 49 malware families and were collected in the period of August 2010 to October 2011. Contagiodump offers an upload dropbox to share mobile malware samples among security researchers and currently hosts 114 items.

[19] http://www.malgenomeproject.org
[20] http://contagiominidump.blogspot.nl

Chapter 3

Design

In this chapter, we define the scope of our analysis framework as it will be outlined in this Chapter 4. We do this by first discussing the framework's design in Section 3.1, followed by a specification of the method trace component in Section 3.2.

3.1 Design

In order to implement a framework for automated analysis of Android applications, we have to come up with a solution for two problems.

1. What kind of information would we would like to collect from the application?
2. How do we run the application in a sandboxed environment and let it execute different control paths to increase the code coverage?

In the following two sections, we first discuss in Section 3.1.1 what kind of information we like to collect, followed by an overview of the framework's design that specifies how applications are sandboxed and simulated in Section 3.1.2.

3.1.1 What to collect

In general, dynamic analysis is used to get an overview of the system calls made by the targeted application as such overview provides a good insight into the app's capabilities. For Android, we would like to do something similar. The Android app's life cycle, however, allows us to extend this concept a bit further.

As outlined in Section 2.2, in contrast to regular binaries seen on desktop PCs, Android applications are Java based. In order to run these apps, the Dalvik Virtual Machine is responsible for interpreting, translating and executing the app's bytecode. This intermediate stationary between application blob and code execution is a perfect place to implement the core of our dynamic analysis platform: by installing specific hooks within the bytecode interpreter, we can display detailed information on an app's internal process (i.e., its function calls and return statements) and thus implement our own method tracer dubbed TraceDroid. In addtion, we can still run target apps while tracing them with the `strace` utility to get an exclusive list of system calls.

The combination of method and system call traces provides detailed information on an app's internal functioning. It could be used by anti-virus analysts to reverse engineer suspicious applications and identify malicious behavior. In addition, software developers could use the extensive method trace output as a debugging tool.

3.1.2 Framework design

To run applications, we use the Android qemu-based emulator that comes with the Android Software Development Kit (SDK). Google already made it easy to deploy a new Android Virtual Device (AVD) and interact with it to trigger specific simulations (initiating phone calls, receiving text messages, etc.). We decided to base our framework on Android version 2.3.4 (codename Gingerbread) as this is currently still on of the most distributed Android versions[1]

We decided to build a Python based framework that accepts an APK file as input and outputs a log directory that holds the dynamic analysis results. The conceptual design for this framework is illustrated in Figure 3.1.

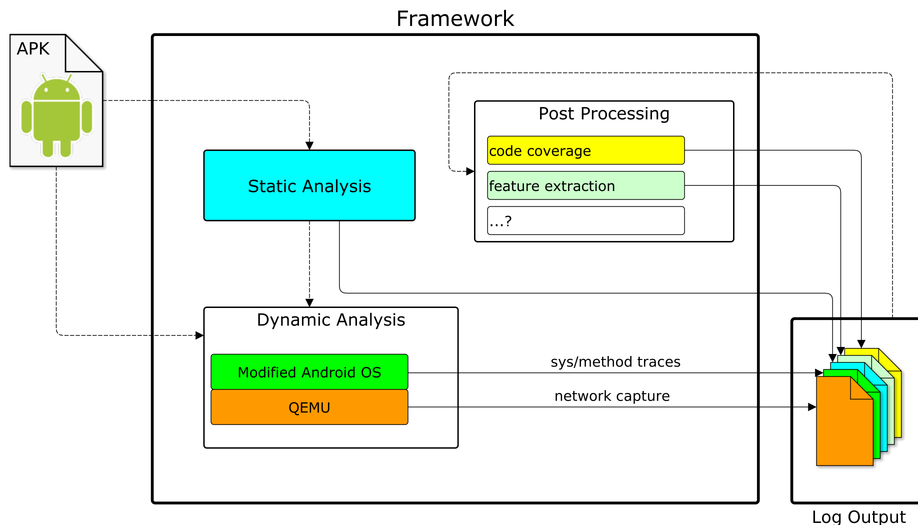


Figure 3.1: Design for Android dynamic analysis platform

As illustrated in Figure 3.1, the app will be installed and executed in a modified Android OS that runs on top of qemu. The OS will have tracing capabilities added that generates method trace output on a per process, per thread basis. In addition, we will enable the capture of network traffic at the level of qemu.

Although the framework focuses on dynamic analysis, some static analysis is neicessary to understand how to simulate different events. Basically, we need to parse the app's `AndroidManifest.xml` file to get information about the app such as its package name and the names of the activities and services it comes with. We make use of the ANDROGUARD [20] project to fetch this information.

[1]`http://developer.android.com/about/dashboards/index.html`

The recently discovered OBad malware sample demonstrated that the manifest file could be corrupted and become unreadable by our static analysis tool [66]. In this event, we still continue dynamic analysis and try to get the required information in a later stage of the analysis flow.

The framework will accept plug-ins that can act as post processing scripts. These scripts receive the location of the log output directory containing the method traces and network traffic capture, as well as the location of the original input APK and may then use these files to compute more interesting analysis results.

As the framework will use TraceDroid as its core component to generate valuable output data, the framework itself is named TraceDroid Analysis Platform (TAP). In this document, the latter is sometimes shorted to simply TraceDroid when its clear that we refer to the platform instead of the method tracer.

3.1.3 Native code

We decided to exclude native code execution from the scope of our work and will thus not discuss it further in much greater detail. The reasons for doing so are manifold. First, although Spreitzenbarth et al. find that the number of applications that use native code is relatively high with 24%, they also conclude that only 13% of the malicious apps make use of native code [64]. Apps that make use of native code are thus not necessarily more likely to be malicious. This observation is probably caused by the fact that essential Android features are not accessible using native code alone, which is our second argument for not focusing on it explicitly. Finally, there are already two effective tools to trace native code: `strace` for system call tracing and `ltrace` to keep track of library invocations. We think that using a combination of these tools provide a detailed enough output trace to study native code execution.

3.2 Specification

In this section, we establish a soft requirement and desired output overview for the method tracer, followed by a short discussion on existing solutions and why they are not sufficient.

3.2.1 Specification

We like TraceDroid to produce readable and easy to understand output files. Ideally, output shall look similar to the original source files of the analyzed application. This is hard to achieve using dynamic analysis alone, as the automated simulation of events may not be able initiate all possible control flow paths, resulting in incomplete output. We would also have to consider loop detection and rewrite `for` and `while` statements, something we think is out of scope for a first version. We decided an overview of all called methods (and API calls in particular) alone would already be of tremendous value for the analyzer. We would like to see all the method calls that an app makes, including the value of the provided parameters and their concluding return statements or thrown

exceptions. In the future, we may then add the tracing of field operations on objects or primitives.

Considering a really simple Android application as depicted in Listing 3.1, we would like to have output similar to Listing 3.2.

Listing 3.1: Source code for a very simple Android app

(a) MainActivity.java

```
package com.vvdveen.example1;

import android.os.Bundle;
import android.app.Activity;

public class MainActivity extends Activity {

 /* Entry point */
 protected void onCreate(Bundle b) {
  super.onCreate(b);

  SimpleClass sc = new
        SimpleClass("new class", 42, 7);

  int min = sc.min();
  System.out.println("minimum: " + min);

  int mul = sc.mul();
  System.out.println("multiplied: " + mul);
 }
}
```

(b) SimpleClass.java

```
package com.vvdveen.example1;

public class SimpleClass {
 String name;
 int i1, i2;

 public SimpleClass(String name,
                       int i1,
                       int i2) {
  this.name = name;
  this.i1 = i1;
  this.i2 = i2;
 }

 public int min() {
  if (i1 < i2) return i1;
  else         return i2;
 }
 public int mul() {
  return i1 * i2;
 }

 public String toString() {
  return this.name;
 }
}
```

As can be derived from Listing 3.2, we want to display a lot of information about the objects and packages that are used. This will come in useful when analyzing large applications that come with many different classes. We also think that displaying parameters and return values will be of high value for the analysis results.

To summarize, TraceDroid should fulfill the following requirements:

- Enable or disable method tracing on a per app basis to avoid a bloat of unrelated trace output for apps running in the background.
- Stick to the bytecode of the target app to avoid a bloat of internal system library calls (we are not interested in the implementation of, for example, `System.out.println()`).
- For each called method, include the name of the class it belongs to.
- For non-static methods, include the `.toString()` result of the corresponding object.
- Print the provided parameters and return values and Call `.toString()` if the value is an object.
- Separate output files per thread to get a better understanding of what is happening when and where.

Listing 3.2: Desired trace output

```
protected void com.vvdveen.example1.MainActivity(<this>).onCreate(<b>)
 protected void android.app.Activity(<this>).onCreate(<b>)
 return
 new com.vvdveen.example1.SimpleClass( (String) "new class", (int) 42, (int) 7)
 return
 public int com.vvdveen.example1.SimpleClass("new class").min()
 return (int) 7
 public void System.out.println("minimum: 7")
 return
 public int com.vvdveen.example1.SimpleClass("new class").mul()
 return (int) 294
 public void System.out.println("multiplied: 294")
 return
return
```

- Include some form of indentation to indicate call depth.
- Add a timestamp to each line.
- Process thrown exceptions correctly (i.e., notice exceptions being forwarded from children to parents).

3.2.2 Existing solutions

The Android OS and its SDK already provide a method tracing and profiling solution that collects detailed information on the executed methods during a profiling session[2]. Although the output seems to be quite complete already, the data does not contain parameter and return values. It is also not possible to start the method tracer right at the start of a new application without modifying the source of the app. On top of that, the Android method tracer is including internal system library to system library method calls, something we would like to omit. Finally, the overhead that is introduced by the Android tracer is quite big (results in Chapter 5) and we aim to find a more efficient solution.

Another existing solution would be the use of JDWP (Java Debug Wire Protocol) and a Java debugger (e.g., `jdb`). For this to work though, we would have find a way to make target applications debuggable, and script the setting and unsetting of breakpoints in `jdb` to still get automated code execution. Using the Java debugger, however, would be a fairly interesting approach to get even more information about the app's internal mechanisms, including field operations.

We decided to extend the existing method tracing and profiling functionality.

[2] `http://developer.android.com/tools/debugging/debugging-tracing.html`

Chapter 4

Implementation

In this chapter, we describe how we implemented our automated framework for dynamic analysis of Android applications. The implementation notes are divided into the following sections.

Method tracer The core of our analysis framework is responsible for generating a complete method trace of the target app. The method tracer was developed by modifying the Dalvik Virtual Machine and is dubbed TraceDroid. A full review of the TraceDroid implementation is detailed in Section 4.1. In this section, we also briefly outline the work done in integrating TraceDroid into the Andrubis platform.

Android framework Aside from the modifications made to the Dalvik Virtual Machine, a small set of changes to the internal Android framework were necessary to successfully integrate the new VM into our analysis framework. These changes are discussed in Section 4.2.

Analysis framework The implementation notes for the TraceDroid Analysis Platform (TAP) that is responsible for starting automated analysis and simulating events are outlined in Section 4.3. In this section, we also describe the post processing plug-ins and the `inspect` tool that allows easy inspection of analysis results.

Bytecode weaving An alternative for TraceDroid that uses bytecode weaving was also developed and its techniques are explained in Section 4.4.

4.1 TraceDroid

In this section, we discuss the implementation of a method tracer for the Android operating system by providing a technical analysis of the source code modifications made to the Dalvik Virtual Machine internals. A benchmark of the resulting method tracer can be found in Chapter 5.

4.1.1 Implementation

By extending the profiling section of the Android Dalvik VM implementation, we were able to obtain log output similar to our desired output as depicted in Listing 3.2 on page 23. Most of the work here involved modifying the existing `dvmMethodTraceAdd()` function in `Profile.c` which is called each time a method is entered or left. This enables us to look up the calling class, the method name and the parameters for each method that gets executed, as well as any return value whenever a method returns.

Start tracing

Since we do not want method traces from the entire Android framework, we need to tell the VM which app to trace. As discussed earlier, each app generally has its own `uid`, which is a perfect value to use as a conditional variable. For this, we modified the Dalvik VM initialization code in two ways.

- The `-uid:<uid>` option is added to the initialization function of the VM. When the emulator is started, one can forward this option to the Zygote process (the parent of all VM instances) by providing the `-prop "dalvik.vm.extra-opts=-uid:<uid>"` argument. It is important to note that the Zygote is only started once, and providing the `uid` parameter is thus only possible during the boot procedure.
  Whenever the Zygote `fork()`s and gains a new `uid`, we check whether it matches the provided `uid` and enable the method tracer in case it does. Note that if an application `fork()`s new processes itself, the `uid` will remain the same. This means that method tracing is enabled automatically for children created by the application.
- A second check is added just after a new VM is `fork()`ed and starts its initialization. We try to read an integer from the file `/sdcard/uid`. If this succeeds, and if it matches the `uid` of the new VM process, we will enable the method tracer. This mechanism can be used to start method tracing an app for which we did not know the `uid` before the emulator was booted.

The `uid` of an app can be found by parsing the `/data/system/packages.list` file. The method tracer is started by calling `dvmMethodTraceStart()`, an existing function which does all the initialization.

We write trace output to `/sdcard/`. However, since VMs are running as ordinary users, they do not have write access to the `/sdcard/` file system by default. This requires a special permission request in the app's `AndroidManifest.xml`. To make sure that we can always write trace files to `/sdcard/`, we modified the initialization code so that new apps are always a member of the `WRITE_EXTERNAL_STORAGE` group.

Profiler control flow

Whenever the original VM's bytecode interpreter enters of leaves a function, the methods `TRACE_METHOD_ENTER`, `TRACE_METHOD_EXIT` and `TRACE_METHOD_UNROLL` (for unrolling exceptions) are called. These functions check for a global boolean `methodTrace.traceEnabled` to be true, and if it is, call `dvmMethodTraceAdd()` which writes trace data to an output file. To extend the method tracer, we

modified the prototypes of these functions so that they expect two extra variables:

- `int type` is used to identify the origin of the call to `TRACE_METHOD_*`. We need this to distinguish specific inlined function calls from regular functions, which we will discuss in more detail later.
- `void *options` is used to store extra options that we need inside the method tracer. For entering an inlined function, the function's parameters will be stored in this pointer as a `u4[4]`. For `TRACE_METHOD_EXIT`, it will contain the return value as a `JValue` pointer and for `TRACE_METHOD_UNROLL`, the thrown exception class is stored in this pointer.

We now describe the control flow inside `dvmMethodTraceAdd()` whenever a target function f is entered.

Initialization First, a check is performed to see if the caller of f is a function from a system library. If this is true, we only continue if f is not a system library function as well to avoid uninteresting method traces. To distinguish system library bytecode from target app bytecode, we introduce a new `boolean isSystem` in the `DvmDex` struct which contains additional VM data structures associated with a DEX file (a filename pointer to the APK or `.jar` filename was added as well for debugging purposes). The value of `isSystem` is set in `dvmJarFileOpen()` in `JarFile.c` whenever the loaded file has a filename that starts with `/system/framework/`.

What follows is a sanity check to make sure that we are not already inside `dvmMethodTraceAdd()`. This may happen when we call `toString()` on objects in a later stage and by doing so we avoid an endless loop. As soon as the test passes, we set `inMethodTraceAdd` to true for the current thread.

Depending on the action we found, we now take a different branch in the tracing code.

Entering a method: `handle_method()` The `handle_method` function is responsible for generating a function entry method trace line. We start with generating the prefix of the output line that consists of a timestamp and some indentation to get readable output. Next, `getModifiers()` generates a list of Java modifiers that are applicable to f (`final`, `native`, `private`, ...). We then get f's return type using `dexProtoGetReturnType()` which returns a type descriptor[1]. We convert the returned type descriptor as well as f's class descriptor to something more readable by using `convertDescriptor()`.

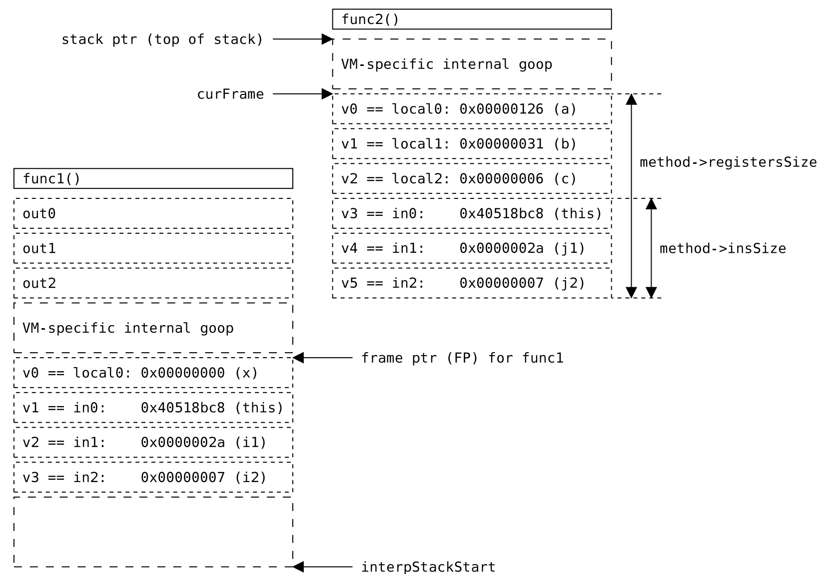
If f is not a constructor call (i.e., `new Object()`), we now generate a string representation of the object. In `getThis()`, we first test if f is static as static methods never have a `this` value. If f is non-static, we call `objectToString()` on the appropriate argument to convert `this` to a string representation. For normal functions, `this` will be the first argument[2].

By adding the offset `method->registersSize - method->insSize` to the current thread's frame pointer we find the reference to the first argument. To understand why this particular offset is used, consider the example source listed in Listing 4.1a and its corresponding stack layout in Figure 4.1b.

[1] `http://source.android.com/tech/dalvik/dex-format.html`
[2] `http://source.android.com/tech/dalvik/dalvik-bytecode.html`

```
public void func2(int j1, int j2) {
 int a, b, c = 0;

 a = j1 * j2;
 b = j1 + j2;
 c = j1 / j2;

 /* Current instruction pointer
  * points here.
  */
}

public void func1(int i1, int i2) {
 int x = 0;

 func2(i1, i2);
}

func1(42, 7);
```

(a) Source code



(b) Stack layout

Figure 4.1: Example stack layout

Although Figure 4.1 shows the stack layout at the moment that `func2()` is about to return, the layout during function entry is the same. The only difference would be that the values of `v0`, `v1` and `v2` were not yet initialized.

Now all that is left is populating the parameters. We generate a string array of parameters in `getParameters()`, followed by constructing a readable string containing these parameters in `getParameterString()`. In `getParameters()`, we loop over the in-arguments of f. We must keep in mind that some functions do not have a `this` reference, which complicates the for loop a bit. We use the `DexParameterIterator` struct and `dexParameterIteratorNextDescriptor()` function to get the corresponding descriptor along with the parameter. For each parameter, we then call `parameterToString()` to convert the parameter to a string.

`parameterToString()` expects two `u4` argument values that represent the parameter: `low` and `high`. `high` will only be used when the parameter is a 64 bit width argument (`doubles` and `longs`). The function also expects a char pointer to the type descriptor of the parameter. The function then performs a simple case/switch statement to construct the correct format string, depending on the descriptor. Up to `void`, all transformations are pretty straightforward. `chars` are a bit more complex due to the fact that Java UTF-16 encoded characters must be converted to printable UTF-8 C strings. For arrays, we simply fall through to the next case, which is the `L` (object) descriptor. Note that we could do a bit more effort here and try to convert arrays of a primitive type to readable output as well. For objects, `objectToString()` is called to convert the reference to a valid C string representation.

`handle_method()` now calls `LOGD_TRACE()` to print the final formatted string to the appropriate file. `LOGD_TRACE()` is an inline function that first locks a dedicated writelock mutex, followed by preparing the output file (if this was

Table 4.1: Possible race condition in LOGD_TRACE

Δt	thread A	thread B
0	Trace is started	...
1		dvmMethodTraceAdd()
2		LOGD_TRACE()
3		fd2 = fopen("outputB", 'a')
4	am profile <pid> stop	
5	fclose(fd1)	
6	fclose(fd2)	
4		fwrite(fd2, ...)

not yet done before) using prep_log(). prep_log() opens a new file in append mode, called dump.<process-id>.<thread-id> in the preset output directory (/sdcard/ or /data/trace/). true is returned if the file is ready for writing, false otherwise (we ran into a couple of samples where fopen() failed since there was no space left on the device). The writelock mutex is used to make sure that there will be no writes when the method tracer is being disabled. An example race condition that we avoid using the writelock mutex is illustrated in Table 4.1

The remaining bits in handle_method() relate to freeing the memory regions that were used to store the output lines. When handle_method() returns, we increase the depth value for this thread so that indentation is setup correctly for the next function entry.

It must be noted here that the TraceDroid performance may be improved by replacing the LOGD_TRACE() calls with a modified version of the log writing function of the existing Android method tracer: *if we're running on the emulator, there's a magic page into which we can put interpreted method information. This allows interpreted methods to show up in the emulator's code traces.* This is an Android modification to the qemu sources to add support for tracing Java method entries/exits. The approach uses a memory-mapped page to enable communication between an application and the emulator[3]. Further research is necessary to figure out how this can be achieved and if there really is a notable performance gain.

Returning from a method: handle_return() When the action given to dvmMethodTraceAdd() equals METHOD_TRACE_EXIT (whenever a return statement is interpreted), and if there is no pending exception, handle_return() will be called to print a return <type> [<value>] trace line. When finished, the depth value for this thread is decreased to setup the indentation correctly for the next function entry. Its implementation is similar to handle_method().

Throwing an exception: handle_throws() When the provided action equals METHOD_TRACE_UNROLL or METHOD_TRACE_EXIT while there is a pending exception, handle_throws() will be called to print a throws <exception> trace line. A pending exception during a METHOD_TRACE_EXIT action indicate that f's parent catches the thrown exception, while the METHOD_TRACE_UNROLL action

[3] http://android.googlesource.com/platform/external/qemu/+/9980bbb9965ee2df42f94aafa817e91835dad406

indicate that the exception will be forwarded to the next parent in line and that intermediate functions are 'unrolling'. The implementation is similar to `handle_method` and `handle_return()`. For unrolling methods, the exception will be stored in the options argument as a `Object*`. For `METHOD_TRACE_EXIT` actions, we fetch the exception our self using `dvmGetException()`. By using this schema, the example source code shown in Listing 4.1a will result in the method trace output as shown in Listing 4.1b.

Listing 4.1: Method trace for thrown exceptions

(a) Source

```
public void f3() throws NullPointerException {
  throw new NullPointerException();
}

public void f2() throws NullPointerException {
  f3();
}

public void f1() {
  try {
    f2();
  } catch (NullPointerException e) {
  }
}
```

(b) Method trace output

```
public void f1()
 public void f2()
  public void f3()
   new java.lang.NullPointerException()
   return (void)
  throws java.lang.NullPointerException
 throws java.lang.NullPointerException
return (void)
```

Inline functions Inline functions require a special approach since their arguments can no longer be fetched from the frame pointer. During a profiling session, `dvmPerformInlineOp4Dbg(u4 arg0, u4 arg1, u4 arg2, u4 arg3)` is responsible for interpreting inlined methods. We modified this function so that it passes an `u4` array to the `TRACE_METHOD_ENTER` prototype that contains the arguments. As outlined earlier, we identify inline methods in `dvmMethodTraceAdd()` by providing a `type` value equal to `TRACE_INLINE`.

The DEX optimization mechanism is in charge for deciding whenever a function shall be inlined or not. In general, we see that many `equals()` calls get inlined.

Stop tracing

Since method trace lines are written to files on disk using `fprintf()`, one needs to explicitly stop the method tracer in order to flush all buffers to disk. During a normal execution flow, method tracing is stopped by executing the `am profile <pid> stop` command, which triggers a call to `dvmMethodTraceStop()`. In here, code is added that loops over the thread list and `fclose()`s any open method trace output file.

Unfortunately, `dvmMethodTraceStop()` is not called when apps run into an uncaught exception. To avoid incomplete log files, we added a similar `fclose()` loop in `threadExitUncaughtException()` which is called whenever a thread runs into such exception. It is not stated that uncaught exceptions will result

in a total VM crash, which is why trace output files may be reopened again in append mode by `prep_log()`.

Added extra VM options

To conclude, below is an overview of added VM options and a short description. VMs will be started with these extra options by providing the `-prop "dalvik.vm.extra-opts=<option1> <option2> ..."` argument to the emulator.

- `-uid:[UID]` Enable method tracing for the app with `uid` equals `UID`.
- `-tracepath:/data/trace` Store trace output files in `/data/trace/` instead of `/sdcard/`. This option can be used if the tracer will be started during boot and `/sdcard/` is not yet mounted. The caller has to make sure that the `/data/trace/` directory is created in order to successfully start tracing.
- `-no-timestamp` Disable timestamps in the method traces. Used for debugging and benchmarking purposes.
- `-no-tostring` Disable `toString()` lookups. Used for debugging and benchmarking purposes.
- `-no-parameters` Disable parameter lookups. Used for debugging and benchmarking purposes.

4.1.2 Andrubis integration

As part of a SysSec[4] scholarship between the Systems Security Group of the VU University, Amsterdam and the SecLab of the Technical University, Vienna, work was done on the integration of TraceDroid into the Anubis/Andrubis platform. Anubis is an online service for analyzing malware, developed by the International Secure Systems Lab[5]. Originally only targeting Windows PE-executables, it was recently extended to accept and analyze Android applications as well, codename Andrubis. The goals of Andrubis are similar to ours, which is why cooperation was an obvious decision.

The Andrubis framework is based on DroidBox [41] for Android 2.1 which was ported by the Andrubis developers to Android 2.3.4 a few months after Andrubis was first released. DroidBox is essentially TaintDroid plus some extra Dalvik VM modifications that log specific API calls. Andrubis uses the modified DroidBox output to generate XML files that contain the analysis results. It also performs a classification algorithm that results in a maliciousness rating between 0 (likely benign) and 10 (likely malicious).

The dynamic analysis results highly depend on the API calls that are tracked by DroidBox, while TraceDroid provides an overview of *all* API calls, plus the functions that are called within the package. On top of that, as we will outline in Section 4.3.3, the TraceDroid output can be used to make statements on the effectiveness of the complete framework. It was thus decided to implement the TraceDroid changes into the existing Android source trunk directory of Andrubis.

[4] `http://www.syssec-project.eu`
[5] `http://www.iseclab.org`

Listing 4.2: Actual trace output

```
1372630874895660: new com.vvdveen.example1.MainActivity()
1372630874937955:  new android.app.Activity()
1372630874938174:  return (void)
1372630874938249: return (void)
1372630874942135: protected void com.vvdveen.example1.MainActivity("com.vvdveen.example1.
                                MainActivity@40516f98").onCreate((android.os.Bundle) "null")
1372630874942666:  protected void android.app.Activity("com.vvdveen.example1.
                                MainActivity@40516f98").onCreate((android.os.Bundle) "null")
1372630874974343:  return (void)
1372630874974504:  public java.lang.Class java.lang.ClassLoader("dalvik.system.PathClassLoader[/data/
 app/com.vvdveen.example1-1.apk]").loadClass((java.lang.String) "com.vvdveen.example1.SimpleClass")
1372630874975984:  return (java.lang.Class) "class com.vvdveen.example1.SimpleClass"
1372630874976467:  public java.lang.Class java.lang.ClassLoader("dalvik.system.PathClassLoader[/data/
 app/com.vvdveen.example1-1.apk]").loadClass((java.lang.String) "java.lang.String")
1372630874976876:  return (java.lang.Class) "class java.lang.String"
1372630875013498:  new com.vvdveen.example1.SimpleClass((java.lang.String) "new class",
                                                        (int) "42", (int) "7")
1372630875013675:  return (void)
1372630875013739:  public int com.vvdveen.example1.SimpleClass("new class").min()
1372630875013836:  return (int) "7"
1372630875013955:  public java.lang.Class java.lang.ClassLoader("dalvik.system.PathClassLoader[/data/
 app/com.vvdveen.example1-1.apk]").loadClass((java.lang.String) "java.lang.System")
1372630875014380:  return (java.lang.Class) "class java.lang.System"
1372630875014793:  public java.lang.Class java.lang.ClassLoader("dalvik.system.PathClassLoader[/data/
 app/com.vvdveen.example1-1.apk]").loadClass((java.lang.String) "java.lang.StringBuilder")
1372630875015190:  return (java.lang.Class) "class java.lang.StringBuilder"
1372630875015477:  new java.lang.StringBuilder((java.lang.String) "minimum: ")
1372630875015692:  return (void)
1372630875015755:  public java.lang.StringBuilder java.lang.StringBuilder("minimum: ")
                                                                     .append((int) "7")
1372630875015938:  return (java.lang.StringBuilder) "minimum: 7"
1372630875016121:  public java.lang.String java.lang.StringBuilder("minimum: 7").toString()
1372630875016277:  return (java.lang.String) "minimum: 7"
1372630875016363:  public void com.android.internal.os.LoggingPrintStream("
 com.android.internal.os.AndroidPrintStream@4050e590").println((java.lang.String) "minimum: 7")
1372630875056811:  return (void)
1372630875056916:  public int com.vvdveen.example1.SimpleClass("new class").mul()
1372630875057051:  return (int) "294"
1372630875057463:  new java.lang.StringBuilder((java.lang.String) "multiplied: ")
1372630875057637:  return (void)
1372630875057700:  public java.lang.StringBuilder java.lang.StringBuilder("multiplied: ")
                                                                     .append((int) "294")
1372630875058035:  return (java.lang.StringBuilder) "multiplied: 294"
1372630875058154:  public java.lang.String java.lang.StringBuilder("multiplied: 294").toString()
1372630875058309:  return (java.lang.String) "multiplied: 294"
1372630875058405:  public void com.android.internal.os.LoggingPrintStream("
 com.android.internal.os.AndroidPrintStream@4050e590").println((java.lang.String) "multiplied: 294")
1372630875059565:  return (void)
1372630875059649: return (void)
```

The TraceDroid patches were implemented into the Andrubis source trunk by Lukas Weichselbaum, the current maintainer of the Andrubis Android sources. The TraceDroid integration into Andrubis is evaluated in more detail in Chapter 5.

4.1.3 Discussion

We have described the necessary steps to extend the existing Android method tracer so that its output includes parameter resolution and return value representation. Looking back to the specification outlined in Section 3.2, the desired method trace output for a given example application in Listing 3.1 and 3.2 on page 22, and the final method trace output as depicted in Listing 4.2 we conclude that TraceDroid successfully implements our requirements.

A full benchmark of TraceDroid can be found in Chapter 5.

4.2 Android Framework Modifications

Although the VM patches discussed in Section 4.1 may be sufficient for simple analysis, some changes had to be applied to Android's internal framework as well to allow better automated analysis support. In this section, we describe the required steps to update the Android framework in order to achieve an optimal integration between TraceDroid and the Android OS.

4.2.1 `killProcess()`

A problem that arises when running automated analysis, is that there are many situations in which the Activity Manager (AM) may decide to kill our target application. On the Android platform, killing an app means killing the corresponding VM, and thus destroying our not-yet-flushed-to-disk method trace data. The best approach to overcome this issue, would be to modify the Android kernel signal handler in such a way that method tracing is stopped before the actual signal is send to the VM. Analysis showed, however, that most kills originate from a single class within the Activity Manager Service, which is why we decided to rather change the AM implementation than to rewrite dangerous kernel code.

Most kills are executed from within `ActivityManagerService.java`. To send the `SIGKILL` signal, the AM calls the static method `killProcess(pid)`. An intermediate method named `killProcess(process, pid)` was added to `ActivityManagerService.java`, which calls `killProcess(pid)` as well, but not before disabling method tracing for the process that is about to get killed. It does so by calling the `stopProfile()` function, which is an existing function used by the `am profile <pid> stop` command. By translating all existing `killProcess(pid)` calls to the new prototype, we make sure that method tracing is stopped before an app is killed, and thus decreasing the number of incomplete log files.

4.2.2 Making `am profile stop` blocking

To ensure that all method trace data is flushed to disk before our analysis platform fetches these files from the virtual android device, we use the existing `am profile <pid> stop` command. This command disables the method tracer for the requested `pid` and results in `fclose()` calls on open log files as explained in Section 4.1.1. The `am profile` command, unfortunately, is non blocking which makes it difficult for the framework to understand when the method trace files are ready to be retrieved.

We modified the internals of the ActivityManager to make the `am` command block until the method tracer is completely disabled and all cached method tracer buffers are flushed to disk. Due to the different layers of abstraction used within the Android OS, it was a cumbersome but interesting process to follow the function calls and understand how and where changes were necessary. We now discuss these changes in a bit more detail and as a result provide an overview of how Android's IPC and its abstraction layers are implemented.

From `am` to AM

Our entry point is the `am` command which is implemented in the `Am.java` source file. In `Am.java`, an `ActivityManagerProxy` to the global activity manager is retrieved using the `ActivityManagerNative.getDefault()` constructor from `ActivityManagerNative.java`. The abstract `ActivityManagerNative` class is extended by `ActivityManagerService`, which is the global activity manager. The interface implemented by `ActivityManagerNative` is `IActivityManager`, which is, according to the source code documentation, a *system private API for talking with the activity manager service. This provides calls from the application back to the activity manager.*

It is now possible for `am` to interact with the activity manager by using the proxy: the `profile <pid> stop` command is implemented by calling the `ActivityManagerProxy.profileControl()` function. This proxy function calls `transact()` which triggers the IPC between `am` and the Activity Manager using the Binder kernel driver. `ActivityManagerNative`'s `onTransact()` will take over control and we now successfully switched from user process `am` to the global activity manager which has information on all running processes.

From AM to target app

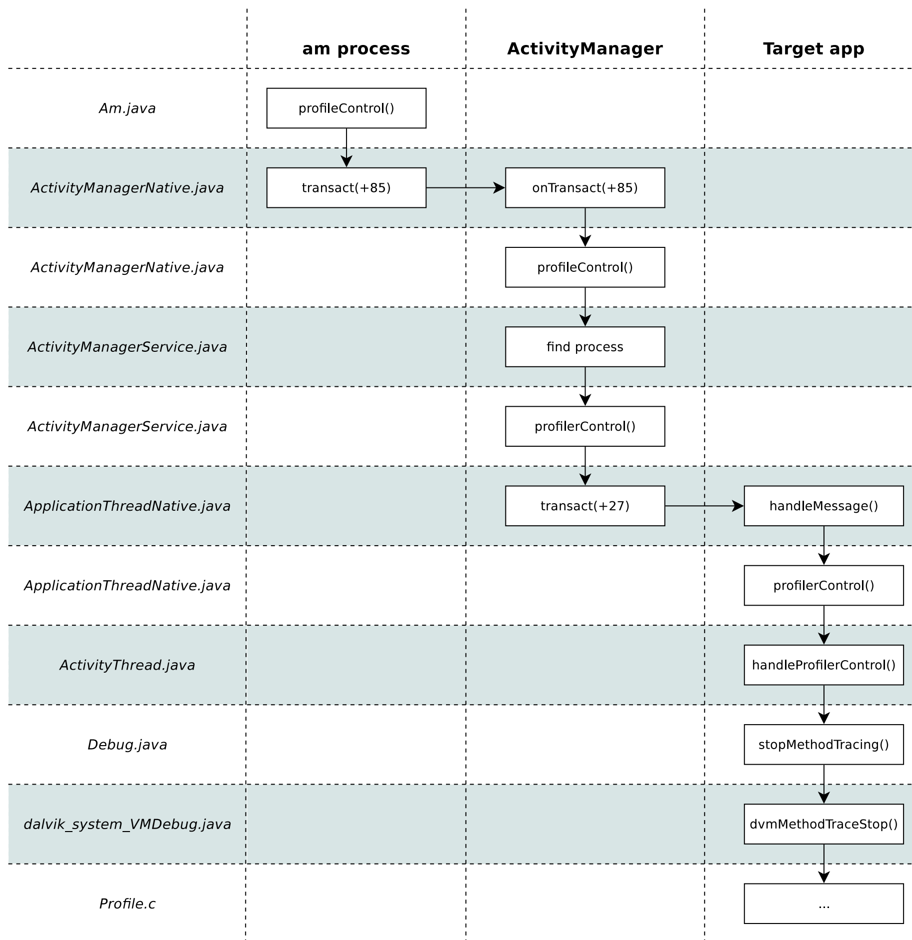
`ActivityManagerNative.onTransact()` calls `profileControl()`, a function implemented in `ActivityManagerService.java` which looks up the requested process in its map of `ProcessRecords` and does some sanity checks. If these pass, `profilerControl()` is called on the main thread of the target process which is implemented in `ApplicationThreadNative.java`.

The abstraction layer used here is similar to one we mentioned earlier: the abstract `ApplicationThreadNative` class is extended in `ActivityThread.java`, while it also has an `ApplicationThreadProxy` class that proxies requests between the Activity Manager and target applications. The `profilerControl()` function of this class makes a call to `transact()` to initiate Binder IPC with the `ActivityThread`. Here, we changed the `FLAG_ONEWAY` parameter to `0` to make this a blocking call.

The `IApplicationThread` that is implemented by `ApplicationThreadNative` is a *system private API for communicating with the application. This is given to the activity manager by an application when it starts up, for the activity manager to tell the application about things it needs to do.*

From app's main to DVM

`ApplicationThreadNative.onTransact()` calls `profilerControl()` which is implemented in `ActivityThread.java`. We patched this function so that in the




```
+85: PROFILE_CONTROL_TRANSACTION  = IBinder.FIRST_CALL_TRANSACTION+85
+27: PROFILER_CONTROL_TRANSACTION = IBinder.FIRST_CALL_TRANSACTION+27
```

Figure 4.2: Android source code control flow diagram for disabling method tracing

case of a stop request, the handler function handleProfilerControl() is called directly, instead of using the queueOrSendMessage() mechanism. This change ensures that the am profile stop request is completed before the function returns.

handleProfilerControl calls Debug.stopMethodTracing() — implemented in Debug.java — which jumps into the VM by calling the native function VMDebug.stopMethodTracing() in dalvik_system_VMDebug.c. From here, the dvmMethodTraceStop() method in Profile.c is called.

The control flow for am profile <pid> stop to stop method tracing a process is shown in Figure 4.2. The figure shows the three different processes involved and their corresponding source code files.

4.2.3 Timeout values

The overhead caused by TraceDroid is likely to have a significant impact on the overall performance of the Android OS. However, as discussed in Chapter 2, the activity manager uses a fixed timeout value before apps are marked unresponsive and the App Not Responding (ANR) dialog is shown. Such ANR message is of no use for our automated analysis platform, and are in fact an annoyance as they may consume valuable analysis time.

We decided to increase the ANR timeout value to 10s to compensate for the overhead introduced by TraceDroid. This was achieved by modifying the `KEY_DISPATCHING_TIMEOUT` constant in `ActivityManagerService.java`,

Next, we changed the behavior of the activity manager in case an unresponsive app was detected by simply killing the app instead of showing the ANR dialog. This was done by modifying the `appNotResponding()` method in `ActivityManagerService.java`.

On top of that, we modified the AM so that unexpected crash dialogs are only shown for a very brief moment before they are dismissed. The `DISSMISS_TIMEOUT` value in `AppErrorDialog.java` was decreased from 5 minutes to 1 second.

4.3 Analysis Framework

The TraceDroid Analysis Platform (TAP) is responsible for the analysis of an Android application. In essence, the platform accepts an APK file as input, analyzes it and then outputs its findings to a log directory.

The entry point of the framework lies in `analyze.py`. After parsing the arguments, it starts static analysis, followed by dynamic analysis and finally it searches for plug-ins to perform post processing on the log output.

4.3.1 Static analysis

Static analysis relies on a modified version of the Androguard project [20]. We search the `AndroidManifest.xml` file for entities outlined in Chapter 2 and write a short summary to a file named `static.log`. Some of these results will be used by the dynamic analysis section.

It must be noted that a recently detected new malware family exploits a previously unknown vulnerability in the way `AndroidManifest.xml` is parsed [66]. This malware sample caused our static analysis to fail. Our framework can continue analysis if this happened though.

4.3.2 Dynamic analysis

Dynamic analysis triggers the start of a fresh virtual Android device that uses our modified system image with TraceDroid installed. The framework then installs the app, starts logcat, starts network capture and enables VM tracing before returning into a main loop.

The main loop iterates over possible simulation actions and executes these. It is possible to provide a subset of simulations as argument for `analyze.py` to speedup the runtime. By default, TAP simulates the actions depicted in Table 4.2.

Table 4.2: Description of default actions simulated during analysis

Action	Description
Boot	Simulate a reboot of the device by broadcasting the `BOOT_COMPLETED` intent.
GPS fix	Simulate a GPS lock.
SMS in	Simulate the receival of an SMS text message.
SMS out	Simulate the sending of an SMS text message.
Call in	Simulate an incoming phone call.
Call out	Simulate an outgoing phone call.
Network disconnect	Simulate a telephony network disconnect.
Low battery	Simulate a low battery.
Package	Simulate the installation, update and removal of another Android application.
Main	Execute the main activity of the app (the activity that is started when a user would start the app directly from the home screen). Create a screenshot after 10 seconds.
Activities	Execute all activities found during static analysis.
Services	Start all services found during static analysis.
Monkey	Run the monkey exerciser* to simulate (pseudo) random input events.

*Monkey exerciser: `http://developer.android.com/tools/help/monkey.html`

A special action named **manual** is available that will drop an `ipython` shell shortly after the target app was installed to allow manual analysis of the application. One could then use the internal `emudroid` object to trigger specific actions (sending an SMS, receiving a call, etc.). It must be noted here that the the monkey exerciser is always started with the same seed value for the pseudorandom number generator (currently set to 1337). This ensures that re-runs of the exerciser generate the same sequence of events. The stream of generated events include clicks, touches and gestures as they could be made by a regular user.

When all required actions are simulated, method tracing is disabled and log files are pulled from the emulated device.

4.3.3 Post processing

The framework searches and runs Python modules found in the post processing directory. It calls the special `post_analysis()` function which expects the path of the target APK, the path of the output log directory and a `StaticAnalysis` object containing static analysis results.

We implemented a couple of post processing tools that add essential functionality to the dynamic analysis platform. We describe these plug-ins in a bit more detail in the remainder of this section.

Coverage

Having a list of methods that were executed during dynamic analysis as a result of the TraceDroid method tracer, we could compute a code coverage value that shows the percentage of APK functions triggered during analysis. We get a list of functions provided by the APK (by doing some static analysis on the package) and then map the dynamically found functions against it. We map functions based on their Java method signature excluding parameter types and modi-

fiers, i.e., on their `<package>.<subpackage>.<classname>.<methodname>` representation.

This concept is realized in the `coverage.py` module. It is worth mentioning that the coverage plug-in distinguishes two types of coverage computation: *conservative* and *naive*. If the latter type is used, all method signatures that match popular external Android library APIs are ignored. Since many apps come with third-party advertisement libraries such as Google's AdMob[6] or AMoBee's Ad SDK[7], and these APIs usually come with many method signatures, we exclude a number of such APIs from coverage computation to get a better indication of the number of methods called that were written by the app authors them self. The current list of excluded libraries is depicted in Table 4.3.

Table 4.3: Excluded libraries for naive code coverage computation

API	**Description**
AMoBee AdSdk	Advertisement library
AdWhirlSDK	Advertisement library
Android API	Official Android API
Android Support API	Official Android support library
GCM	Google Cloud Messaging library
Google AdMob	Advertisement library
Millennial Media Adview	Advertisement library
Mobclix	Advertisement library
MobFox SDK	Advertisement library
Netty	Network application framework library

Methods are excluded from code coverage computation if their signature matches one of the signatures found in the excluded APIs. By doing so, we take the risk to lose signatures that are part of the app's core packages, but are named according to one of the popular APIs. For this reason, the conservative option was kept as default computation approach.

From now on, naive code coverage computation is used unless stated otherwise.

Feature extraction

The list of called methods, along with their parameter and return values are used by our second plug-in to generate a feature footprint of the target app. The resulting feature set may be used by a machine learning algorithm to cluster and classify new apps and maybe even identify them as being malicious or benign. The feature sets that we currently track is depicted in Table 4.4. On top of that, bloom vectors are generated to store information about the (number of) API calls.

Database storage

This simple plug-in writes log results to a `sqlite3` database so that they can be used in a later stage for automated analysis of the analysis results.

[6] `http://www.google.com/ads/admob`
[7] `http://www.amobee.com/technology/ad_sdk.shtml`

Table 4.4: Description of different feature sets extracted

Feature set	Description
telephony.*	To indicate the retrieval of core telephony records (IMEI, IMSI, MSISDN, etc.).
sms.*	To indicate SMS message activity (sending, parsing).
content.*	To indicate special android.content.* method invocations (getSystemService(), startService(), pm.signature.*, etc.).
io.*	To indicate input/output activity (database I/O, file open/read/write, etc.).
misc.*	To indicate a number of miscellaneous activity (crypto operations, digest operations, native function calls, reflected methods, zip operations, etc.).
network.*	To indicate network operations.

4.3.4 Inspecting TraceDroid output

The framework comes with a parser named `trace.py` to parse trace files generated by TraceDroid. The parser loads function or constructor entries and their corresponding return values or thrown exceptions into single Python objects before dropping an `ipython` shell for easy access.

In this section, we describe the `trace.py` functionality in more detail. The inspect tool is demonstrated in more detail in Chapter 5, where we analyze a real-world Android malware example.

Description of fields

The main function of `trace.py` searches a given log directory for trace files and parses these. An analyst can then access the trace results via the `traces` dictionary which has `(pid, tid)` keys with `Trace` object values. `Trace` objects, on their turn, have a list of found `Function` objects stored in the `functions` field and a list of `Constructor` objects in the `constructors` variable. `Function` and `Constructor` objects come with a number of fields of which most are described in Table 4.5.

Table 4.5: Common fields for `Function` and `Constructor` objects

Field	Description
modifiers	List of modifiers (public, private, static, ...).
parameters	List of parameters represented as (type, value) tuples.
target_object	Full class name (including package name) of the related object.
target_object_s	String representation of the related object in case of an instance method.
exception	Exception thrown (if any).
return_type	Return type for this function.
return_value	String representation of the returned value.
retway	return for return statements or thrown for exceptions thrown.
name	Name of the method.
is_api	Whether or not this is an API call.
depth	Current call depth.
linenumber_{enter|leave}	Corresponding line number in the original trace file.
timestamp_{enter|leave}	Timestamp.
reflected_method	For java.lang.reflect.Method.invoke() calls, this value contains another (stripped) Function object for the invoked method.

For convenience, a couple of helper lists are generated for easy, direct access. These variables are depicted in Table 4.6.

Table 4.6: Variables for direct access

Variable	Description
functions	List of all found functions.
constructors	List of all found constructors.
reflected	List of reflected functions.
fnames	Dictionary of full function names, whether they are API functions and their call count.
cnames	Dictionary of full constructor names, whether they are API constructors and their call count.
rnames	Dictionary of full reflected function names, whether they are API functions and their call count.

The helper function print_names() accepts one of the fnames, cnames or rnames variables and prints a table containing the function names and the number of times they were called.

Call graphs

The parser comes with a generate_callgraph() function that constructs a call graph for the analyzed app using pydot, the Python interface to Graphviz's Dot language. It's parameters are outlined in Table 4.7.

Table 4.7: Options for generate_callgraph()

Parameter	**Value** (default underlined)	Description
apis	True or False	Include API calls in the call graph.
use_clusters	True or False	Group functions that belong to the same class into a subgraph.
use_colors	True or False	Give each class a different color
vertical	True or False	Vertically align nodes in clusters.
splines	'spline', 'ortho' or 'line'	The graphviz spline type that should be used.

The value returned by generate_callgraph() is a pydot.Dot object and may be written to disk using one of the available write_*(<filename>) functions. For example, to write a pdf, use <ret-value>.write_pdf('filename.pdf') or <ret-value>.write_png('filename.png') to write a png. It is also possible to get a more textual representation of the call graph by simply calling print_callgraph() with no further parameters.

4.4 Bytecode Weaving

Bytecode weaving is a technique that combines existing Java bytecode with new code snippets or so-called aspects and is used in the Aspect Oriented Programming (AOP) paradigm. In this section, we outline how Aspect Oriented Programming can help us to write a TraceDroid alternative that works entirely on the application level.

4.4.1 AOP: Aspect Oriented Programming

Using AOP, it is possible to weave new functionality into the existing bytecode of an application without the need of having access to the original Java sources. If we use special trace aspects, we can use this technique to add method tracing functionality into an existing application.

Looking at the previously used `SimpleClass` source (depicted in Listing 3.1b on page 22) and a new Java main source file depicted in Listing 4.3a, we now outline how method tracing can be enabled for this application. In Listing 4.3b, we illustrate how the sources are compiled and how the application is run without any further modifications.

Listing 4.3: Enabling method tracing using AOP

(a) Main.java

```
public class Main {
 public static void main(String[] args) {
  SimpleClass sc = new
        SimpleClass("new class", 42, 7);

  int min = sc.min();
  System.out.println("minimum: " + min);

  int mul = sc.mul();
  System.out.println("multiplied: " + mul);
 }
}
```

(b) Compiling

```
javac SimpleClass.java -d ./classes/
javac Main.java -d ./classes/ -cp ./classes/
rm *.java
java -cp ./classes/ Main

minimum: 7
multiplied: 294
```

(c) Weaving

```
ajc Tracer.aj -outjar Tracer.jar \
                -cp /path/to/aspectjrt.jar \
                -source 1.5
ajc -inpath ./classes/ \
      -aspectpath Tracer.jar \
      -outjar Main.jar
java -cp Main.jar:Tracer.jar:\
           /path/to/aspectjrt.jar Main
```

Using the simplified tracing aspect outlined in Listing 4.4, we add tracing functionality to the existing app in Listing 4.3c using the AspectJ compiler[8]. It must be noted that by using the `call` pointcut (a set of specifications of when the aspect code should be executed), we ensure that the tracing aspect is added to the method that calls the target method. If we would have used the `execution` pointcut, `ajc` will rewrite the prologue and epilogue of the target methods, in which case we miss API calls as the compiler cannot rewrite those methods [40].

If we execute the new `jar` file, we get output as depicted in Listing 4.5 on page 43. It must be noted that this is a simplified working example and some core functionality is still missing (parameter resolution, call depth tracking, etc.), but it is clear that these are trivial to implement now.

A problem that arises when porting this mechanism to Android applications is that existing bytecode manipulation libraries (e.g., Apache's Byte Code Engineering Library (BCEL)[9] or ASM[10]) used by AOP compilers like AspectJ do

[8] `http://eclipse.org/aspectj`
[9] `http://commons.apache.org/proper/commons-bcel`
[10] `http://asm.ow2.org`

Listing 4.4: Minimal method tracing aspect

```
import org.aspectj.lang.reflect.MethodSignature;
import org.aspectj.lang.reflect.CodeSignature;
import org.aspectj.lang.JoinPoint;
import java.lang.reflect.Modifier;

aspect Trace {
 pointcut traceMethods():  call (* *(..)) &&
                          !call (* java.lang.Object.clone() ) &&
                          !cflow(within(Trace));

 before(): traceMethods() {
   entrance(thisJoinPoint);
 }
 after() returning (Object retval): traceMethods() {
     leaving(thisJoinPoint, retval);
 }

 synchronized void entrance(JoinPoint tjp) {
  Object           target =                  tjp.getTarget();
  Object[]      parameters =                 tjp.getArgs();
  CodeSignature signature = (CodeSignature) tjp.getSignature();
  MethodSignature methodSignature = (MethodSignature) signature;

  String log = Modifier.toString(signature.getModifiers()) + " ";
  log += methodSignature.getReturnType().getName() + " ";
  log += signature.getDeclaringTypeName();
  if (target != null) log += "(" + target.toString() + ").";
  else                log += "( null ).";
  System.out.println(log + signature.getName() + "(...)");
 }
 synchronized void leaving(JoinPoint tjp, Object retval) {
  if (retval != null) System.out.println("return " + retval.toString());
  else                System.out.println("return");
 }
}
```

not (yet) support manipulation of Dalvik bytecode. We overcome this issue by converting Dalvik bytecode back to Java classes first, before weaving the method tracer into the bytecode. When weaving succeeded, the patched Java classes are repackaged into a new APK that can be installed on any Android device. For decompiling the Dalvik bytecode to Java bytecode, we use the DEX2JAR tool [70] that converts a given `classes.dex` into a `classes.jar` file. Figure 4.3 depicts the different tools and processes that are involved when we add tracing aspects to existing Android applications.

During analysis, we found that in some cases, the AspectJ compiler crashes during the weaving process of the Java `.jar`. In most cases this was caused by a *jump is too far* error, meaning that adding trace bytecode to the current method exceeds its maximum size (which is limited to 64K[11]). Although a normal app should never come close to the 64K boundary, Java obfuscators like ProGuard[12] may combine or unroll multiple methods into a single large one that comes close to the limit. When trace aspects are weaved into such large methods, it is likely

[11] `http://bugs.sun.com/view_bug.do?bug_id=4262078`
[12] `http://developer.android.com/tools/help/proguard.html`

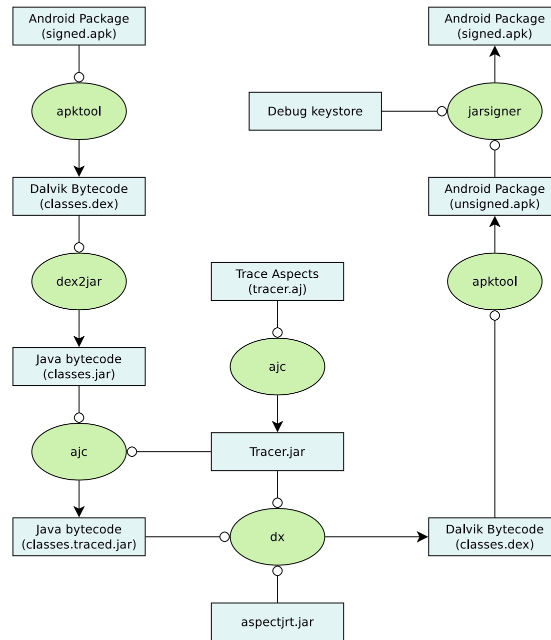


Figure 4.3: Weave process

that *jump is too far* errors arise.

4.4.2 Advantages and drawbacks of bytecode weaving

The VM method tracing technique discussed in Section 4.1 and the rewriting technique discussed here both have their advantages and disadvantages. In general, the VM method tracer yields better results and is more sophisticated than the bytecode weaving approach, while the latter has the advantage that it does not require a modified Android environment to perform the actual analysis. Consider the following summary of package rewriting issues (or disadvantages).

- When rewriting apps, we change the signature of the package's codebase which could be detected by a malicious application. A malware author could decide to not trigger detrimental activities if he finds that the MD5 hash of its classes changed, causing the app to hide its malicious behavior and resulting in a low maliciousness rating.
  Since Android APK containers are essentially signed `jar` files, repackaging means that the signature of the app's author will be replaced with another signature. This could also be detected from within the app, possibly initiating a different control flow for repackaged applications.
  Signature checking could possible be intercepted by special aspects, but that definitely complicates the process.
- Obfuscation techniques could break the repacking process if the the maximum method size is exceeded. To solve this, one would have to use bytecode manipulation to split large methods into smaller chunks (method outlining).

Listing 4.5: Trace aspect output

```
java -cp Main.jar:/path/to/aspectjrt.jar:Tracer.jar Main

public int SimpleClass(new class).min(...)
return 7
public java.lang.StringBuilder java.lang.StringBuilder().append(...)
return minimum:
public java.lang.StringBuilder java.lang.StringBuilder(minimum: ).append(...)
return minimum: 7
public java.lang.String java.lang.StringBuilder(minimum: 7).toString(...)
return minimum: 7
public void java.io.PrintStream(java.io.PrintStream@58fe64b9).println(...)
minimum: 7
return
public int SimpleClass(new class).mul(...)
return 294
public java.lang.StringBuilder java.lang.StringBuilder().append(...)
return multiplied:
public java.lang.StringBuilder java.lang.StringBuilder(multiplied: ).append(...)
return multiplied: 294
public java.lang.String java.lang.StringBuilder(multiplied: 294).toString(...)
return multiplied: 294
public void java.io.PrintStream(java.io.PrintStream@58fe64b9).println(...)
multiplied: 294
return
```

- The weaving process as it is described in this section depends on an external utility to convert Dalvik bytecode to Java bytecode. If an error is found in DEX2JAR and exploited by a malware sample (as with [4]), the weaving process fails.
  It must be noted that DroidBox 2.3 comes with a rewriting module that manipulates Dalvik bytecode directly, and thus circumventing this issue. It is unclear how much effort is needed to extend DroidBox 2.3 to a full method tracer.

It is obvious that TraceDroid does not suffer from above issues. On the other hand, bytecode weaving has the following advantages over TraceDroid.

- By injecting method tracing code snippets into the app directly, there is no longer the need to have a modified version of the Android OS. This makes the analysis platform in which apps are installed and simulated version independent.
- Having method tracing code at the application level, apps can be installed and analyzed on real hardware without the need of building and installing a customized firmware. This makes it possible for end users to analyze applications using their own devices. This has as extra advantage that emulator detection mechanisms used by malware authors become worthless.
- When AspectJ traces are used, as described in this section, extending the tracer becomes as easy as writing a Java module. Implementing array resolution using aspects, for example, will be less complicated using aspects than performing the resolution on a VM level.

Future work may focus on adding support for Dalvik bytecode to the popular bytecode manipulation libraries (BCEL for Dalvik), as well as adding automatic method outlining to the AspectJ `ajc` compiler.

Chapter 5

Evaluation

In this chapter, we evaluate our framework for dynamic analysis of Android applications as it was described in Chapter 4. The contents are divided into the following sections.

Benchmarks. We start with evaluating benchmarking results in Section 5.1.

Andrubis. We show in Section 5.2 that TraceDroid can safely be integrated into Andrubis without losing analysis results that may be caused by the additional slowdown.

Coverages. In Section 5.3, we dive deeper into the analysis framework and describe our test setup for analyzing 500 Android applications. We discuss the effectiveness of our analysis framework by looking at achieved code coverage. We outline results for both TraceDroid and the current Andrubis implementation.

Failures. In Section 5.4, we list outstanding issues found in TraceDroid in combination with the analysis frameworks.

Example. Finally, we analyze an Android malware application in more detail in Section 5.5.

5.1 Benchmarking TraceDroid

In this section, we describe the benchmarking setup (Section 5.1.1) and results (Section 5.1.2) for TraceDroid.

5.1.1 Benchmark setup

To compute the overhead that is introduced by TraceDroid, we updated the Android browser application so that timestamps are printed during the process of loading a webpage. The patch enables logging of timestamps during the following stages of loading a webpage:

1. In the browser's **constructor** code (i.e., when the browser is first started)
2. In the browser's `onPageStarted()` method (i.e., when the browser is about to start loading a webpage). We call this the *ready* time (thread is ready to start loading).

3. In the browser's **`onPageFinished()`** method (i.e., when the browser completed loading a webpage). We call this the *load* time (thread loaded the page).

We use the Java `SystemClock.currentThreadTimeMillis()` function for generating timestamps. By using a clock that provides thread based interval timing, we exclude abnormalities that are caused by background processes. In order to avoid any latency caused by Internet communication, we cached popular websites and loaded them from the phone's SD card. We also made sure that the same Dalvik execution mode (portable interpreter without JIT) was used for each test run to produce fair results.

Each webpage was loaded 10 times before the average ready and load times were computed. The emulator was rebooted between each page load to avoid any cache optimizations. We finally used a number of different speedup setups (using the flags described in Section 4.1.1) to get a better understanding of the expected slowdown:

Baseline Method tracing disabled.
Setup1 Method tracing with `-no-timestamp`, `-no-tostring` and `-no-parameters`.
Setup2 Method tracing with `-no-timestamp` and `-no-tostring`.
Setup3 Method tracing with `-no-timestamp`.
Full Method tracing without any speedup flag.
Android Method tracing using the original Android profiler.

The benchmark test was repeated with a special TraceDroid version where `LOGD_TRACE()` was disabled, thus omitting `fprintf()` calls. Results for this second test would give an indication of the possible performance gain if we could avoid calls to `fprintf()` in favor of using the 'magic' memory-mapped page setup for direct communication between the application and emulator (as discussed in Section 4.1.1 on page 28.

We used the Firefox *Save Page As...* option to generate a cached version of the main pages of the following websites, including shown pictures and external scripts.

- `http://www.google.com`
- `http://www.youtube.com`
- `http://www.amazon.com`
- `http://www.wikipedia.org`
- `http://www.ebay.com`
- `http://slashdot.org`
- `http://stackoverflow.com`
- `http://www.tuwien.ac.at`

In addition, we also browsed to `about:blank`.

5.1.2 Benchmark results

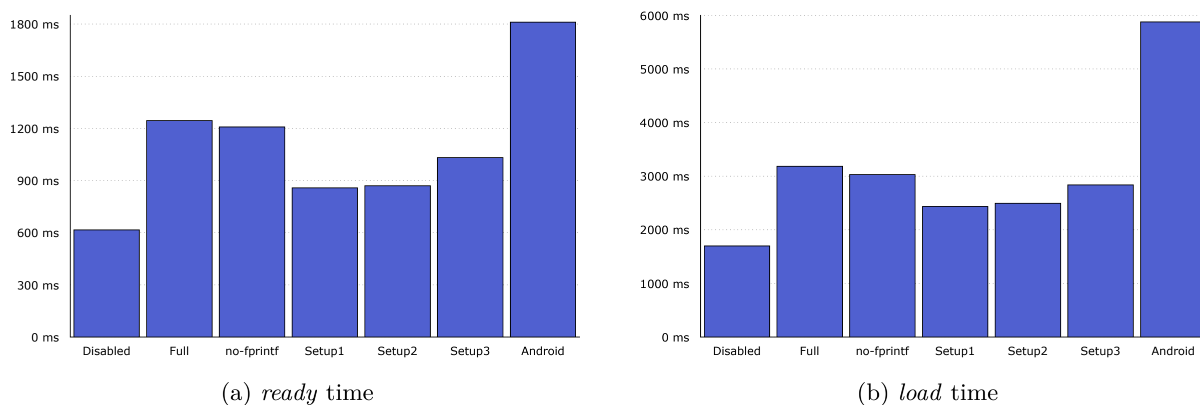
Results for the benchmark tests are depicted in Table 5.1. Timing values are listed in milliseconds. The results are graphically displayed in Figure 5.1.

Studying the results, we conclude that method tracing using TraceDroid introduces an interpreter slowdown of about 100%. The possible 'magic page'

Table 5.1: Benchmark results (all times in ms)

Benchmark	Baseline	Setup1	Setup2	Setup3	Full	Android
Ready	616	858 (+39.29%)	870 (+41.23%)	1032 (+67.61%)	1245 (+102.19%)	1811 (+192.57)%
Ready (`no-fprintf`)	614	825 (+34.45%)	840 (+36.89%)	988 (+60.99%)	1208 (+ 96.74%)	
Load	1699	2434 (+43.29%)	2494 (+46.81%)	2838 (+67.02%)	3185 (+ 87.48%)	5877 (+245.50)%
Load (`no-fprintf`)	1681	2414 (+43.55%)	2419 (+43.85%)	2728 (+62.24%)	3030 (+ 80.19%)	

Baseline: Method tracing disabled
Setup1: -no-timestamp -no-tostring -no-parameters
Setup2: -no-timestamp -no-tostring
Setup3: -no-timestamp
Full: Method tracing without any speedup flag
Android: Method tracing using the original Android profiler



(a) *ready* time

(b) *load* time

Figure 5.1: Benchmark results

improvement discussed earlier will have a very limited effect on the overall performance (about 6%), which is likely due to the fact that the Android OS already does useful buffering during `fprintf()` calls. Object resolution (calling expensive `toString()` methods) and timestamp generation (system calls to `gettimeofday()`) are by far TraceDroid's largest performance killers: the first are responsible for an extra overhead of 20–25%, while the latter introduce another additional 20–30% slowdown.

It is interesting to notice the performance differences between the *load* and *ready* benchmark (15% for the fully enabled method tracer). This indicates that there are more background operations being processed during the *load* than during the *ready* benchmark. These are internal function calls that occur within the Android framework and are not invoked from the Browser's bytecode directly. The bottleneck is the WebKit layout engine that has to render the requested webpage and any JavaScript that comes with it in order to display it correctly. By not tracing the internal WebKit methods, we gain a clear performance boost against the *load* benchmark: the increase of the tracer's workload is less intense than the considerable amount of extra work for the browser.

Finally, compared to the existing Android method tracer, we conclude that a full version of TraceDroid is about 1.45 to 1.85 times faster. This is caused

by the fact that we are not tracking method calls that are inside the Android framework itself. However, if considering the core purpose of method profiling, we feel that omitting these internal calls is more of an advantage than a drawback: app developers (the core group of profiling users) have no control over the internal API implementations anyway, and are likely to rather not have them included in method trace output than to have to filter them out.

5.2 Benchmarking TraceDroid + Andrubis

To decide whether TraceDroid could be integrated into Andrubis without having its performance overhead causing a drop in the number of detected operations, we setup a special Andrubis benchmark test. In this section, we first describe Andrubis in a bit more detail in Section 5.2.1. We then describe our benchmark setup and conclusions in Section 5.2.2.

5.2.1 Andrubis background

As discussed in Section 4.1.2, Andrubis uses a modified DroidBox 2.1 setup combined with TaintDroid to track interesting API calls and specific personal data leaks. Depending on the activities detected during analysis, Andrubis generates a report that contains a number of different operation sections. The operations that are currently being detected by Andrubis are described in Table 5.2.

Table 5.2: Overview of operations detected by Andrubis

Operation group	**Subsections**	**Description**
File operations	**file read**	Reading file contents.
	file write	Writing file contents.
Network operations	**network open**	Opening a network socket.
	network read	Reading from a socket.
	network write	Writing to a socket.
Broadcast receivers	-	A list of (dynamically) installed broadcast receivers.
Data leaks	**network leak**	Leaking personal data via network traffic.
	file leak	Leaking personal data via file writes.
	sms leak	Leaking personal data via SMS texts.
Crypto operations	**crypto key**	Initializing a cryptographic key.
	crypto operation	Cryptographic operations (encrypt/decrypt).
DEX classes loaded	-	A list of dynamically loaded DEX classes.
Native libraries loaded	-	A list of native libraries loaded.
Bypassed permissions	-	A list of permissions bypassed by using another app's capabilities.
Sent SMS	-	SMS text messages sent.
Phone calls	-	Phone calls made.
Started services	-	A list of started services.

Most operations described in Table 5.2 come with a number of different fields. A **file read** operation, for example, would have two fields: `path` for indicating which file was read, and `data` to list the exact stream of bytes read from the file. As another example, **network read/write operations** come with three fields: `host` for storing the targeted host IP address, `port` for storing the used port number and `data` for any data that was written over this connection. Currently, Andrubis does not keep track of the protocol used (TCP/UDP).

It is important to mention that the monkey exerciser setup used by ANDRUBIS was truly random, in contrast to the setup used for TAP. This makes it sometimes hard to compare the output of two ANDRUBIS analysis runs, as the randomness introduces an unpredictable difference between two consecutive monkey runs. This behavior was changed and ANDRUBIS now also uses a constant seed, but these changes were not yet in place during the evaluation of our work.

Since ANDRUBIS runs samples for a small amount of time only (180 seconds), we need to understand whether the TRACEDROID performance overhead will have a negative impact on the operations that are reported by ANDRUBIS.

5.2.2 Benchmark results

We performed automated analysis twice on a small set of applications while method tracing was disabled. We then computed a similarity percentage between the generated reports B_1 and B_2 to establish a baseline difference ΔB. Subsequently, we repeated analysis four times with method tracing enabled while increasing the runtime window and computed the similarity percentage between the generated reports $R_1 \ldots R_4$ and baseline report B_1 (ground truth). By comparing $\Delta R_1 \ldots \Delta R_4$ with ΔB, we can make statements on the impact of TRACEDROID on ANDRUBIS.

A report B contains a number of operations B_{o1}, B_{o2}, ..., B_{on}. Each operation is represented as a tuple `(opcode, non-data fields, data field)` where `opcode` is a combination of the operation group and subsection as described in Table 5.2. Two operations B_{oi} and B_{oj} are stated to be similar when their opcode and all non-data fields are equal. We search R for similar operations to B_{o1}, B_{o2}, ..., B_{on} without allowing repetition. This means that if operation R_{oj} is matched against B_{oi}, it cannot be reused to also match similar operation B_{oj+x}. The similarity value is then computed by dividing the number of matched operations by the number of total operations found in B.

The results are depicted in Table 5.3.

Table 5.3: ANDRUBIS similarities for different runtime values (without repetition)

Set	ΔB	ΔR_1 **(180s)**	ΔR_2 **(240s)**	ΔR_3 **(300s)**	ΔR_4 **(360s)**
17x benign	85.40%	58.72%	76.04%	81.35%	86.48%
18x malicious	91.33%	73.10%	84.51%	89.38%	87.19%

From Table 5.3, we conclude that due to the 100% overhead caused by TRACEDROID, we need to double the ANDRUBIS runtime window in order to maintain the same number of found operations.

It is interesting to notice that a longer runtime window does not necessarily lead to a higher similarity percentage. After analyzing the malicious subset for 300 seconds, a similarity of 89% was achieved, while only 87% of the operations were detected during the 360 seconds run. This is caused by the truly random monkey exerciser setup used in ANDRUBIS.

Before blindly increasing the ANDRUBIS runtime window, we computed the average code coverage gained during dynamic analysis. Due to repeating oper-

ations, we expect that code coverage does not follow the same increase as the number of operations do. The results are depicted in Table 5.4.

Table 5.4: ANDRUBIS coverage results for different runtime values

Set	**180s**	**240s**	**300s**	**360s**
17x benign	30.10%	32.88%	34.01%	34.39%
18x malicious	23.01%	24.56%	25.67%	25.69%

Regarding Table 5.4, We indeed notice that a doubled runtime window does not result in twice the amount of code being executed. Looking at individual sample results, we even see that for many samples code coverage remains almost the same while the runtime value is increased.

This observation of a higher increase of found operations than code coverage is likely caused by loops in the application's codebase. Functions called within a loop that initiates new operations do not increase the code coverage. Another cause are functions called with different parameters or functions called from multiple classes and thus generating multiple operations. These functions are only counted once regarding code coverage.

We repeated the similarities benchmark while removing the requirement that the amount of operations found with method tracing enabled shall be equal or higher than the amount of similar operations found during the baseline test. However, we now do require that two operation's data fields are equal. The results for this second similarities test are illustrated in Table 5.5.

Table 5.5: ANDRUBIS similarities for different runtime values (equal data field required)

Set	ΔB	ΔR_1 **(180s)**	ΔR_2 **(240s)**	ΔR_3 **(300s)**	ΔR_4 **(360s)**
17x benign	75.73%	67.47%	74.79%	71.56%	72.38%
18x malicious	92.24%	84.51%	88.34%	88.74%	88.75%

Studying Table 5.5, we again see the negative effects that the randomness of the monkey exerciser has on the operations initiated by the samples. During the 240 second analysis run, some of the benign samples were triggered in a similar fashion as during the second baseline test, resulting in a small similarity difference ($< 1\%$), while subsequent runs did slightly worse again.

From Tables 5.4 and 5.5 we conclude that by increasing the ANDRUBIS timeout value from 180 seconds to 240 seconds, we find an optimal balance between analysis processing speed and code coverage. Assuming that plain ANDRUBIS was gaining a code coverage of 34.39% for our benign set and 25.69% for our malicious set (the code coverages for a doubled runtime), we only lose about 1 to 2 percent code coverage, while the runtime increase is 33%. In exchange, ANDRUBIS now has full method tracer capabilities which are very convenient for dynamic analysis. Considering that the DROIDBOX additions can now be removed (as DROIDBOX is tracking only a subset of API calls, while we are tracing every API call and more), the overhead can be reduced even further.

5.3 Coverage

In this section, we evaluate the effectiveness of the two dynamic analysis platforms discussed previously (ANDRUBIS and TAP) by looking at the code that was covered during dynamic analysis. In Section 5.3.1, we first compare automated results against manual analysis, followed by a breakdown of different simulation techniques in Section 5.3.2. We conclude with an extensive evaluation of 500 Android applications in Section 5.3.3.

The samples used for analysis consist of a set of 250 malicious and 250 benign samples as selected by the ANDRUBIS team. Unfortunately, the malicious set contained a couple of non-functioning samples, so we were able to use a set of 242 malicious and 250 benign Android applications to run our tests on. A list of used samples can be found in Appendix A.

5.3.1 Compared to manual analysis

In order to use code coverage as a measuring technique for our analysis framework, we first set our expectations by running manual analysis on a subset of samples. By comparing the code coverage achieved during manual analysis against the code coverage gained during automated analysis, we can make statements on the effectiveness of the used simulations techniques. For this, we use the code coverage methodology as outlined in Section 4.3.3 on page 36.

We randomly picked 20 malicious and 20 benign samples. We used a small script that installed each app on a freshly emulated Android platform before giving us a 180 seconds runtime window for our manual stimulation. After 180 seconds (the default ANDRUBIS runtime), the app was closed automatically and code coverage was computed. Within these 180 seconds, we tried to activate as many components of the app as possible. We then analyzed the app automatically using both ANDRUBIS and TRACEDROID while the runtime window of three minutes remained the same.

In Table 5.6, we display the achieved code coverage for all samples that were successfully tested. Although TAP does not use a fixed runtime for each sample, we also include those results. Note that ANDRUBIS failed to perform analysis for 5 samples which were hence omitted from the table. Studying the results, we can make the following observations.

- Despite the fact that naive coverage computation was used, coverage results are still fairly low. We try to explain possible causes for this later in this section.
- The analysis platforms seem to perform better on malicious samples. This is likely caused by the external simulations (e.g., simulate a reboot or receive an SMS text message) that were not triggered during manual analysis. In general, malicious applications are more likely to act upon these events than benign apps, as it allows a malware writer to automatically start intruding background services whenever such event occurs. This could also explain why code coverage for our malicious set is about 10% less than for benign samples. We test this hypothesis in Section 5.3.3.
- There is a large fluctuation between the number of functions that are declared by an app (ranging from 2 to 5813 for this small subset). A closer look at samples with such low amount of methods teaches us that

Table 5.6: Coverage results for benign and malicious samples

MD5 hash	Manual	Calls of total	Andrubis	TraceDroid
03aaf04fa886b76303114bc430c1e32c	34.52%	107 of 310	-4.19%	-4.52%
128a971ff90638fd7fc7afca31dca16b	100.00%	2 of 2	0.00%	0.00%
12b7a4873a2adbd7d4b89eb17d57e3aa	7.55%	216 of 2860	-0.80%	+0.03%
12dc6496fdd54a9df28d991073f26749	35.24%	160 of 454	-32.38%	-2.42%
1390e4fecca9888cdf0489c5fe717839	24.43%	472 of 1932	-5.33%	-1.19%
37eacdc7366403eac3970124c3a3fc32	37.70%	184 of 488	-17.83%	0.00%
3a11d47f994ec85cfeff8e159de46c54	24.08%	657 of 2728	-3.45%	0.00%
f240abe83b8da844f5dfdaceba9a6f7e	31.47%	772 of 2453	-12.27%	-18.55%
f2c3afe177ef70720031f2fb0d0aa343	8.91%	27 of 303	+1.32%	-0.66%
f40759b74eff6b09ae53a0dbcabc07d4	20.90%	98 of 469	-0.64%	-0.43%
f5d6b6b019949329ef0de89aca6ac67e	58.14%	125 of 215	-27.91%	-9.30%
f6a0e9573810d3da8a292b49940b09e2	100.00%	3 of 3	0.00%	0.00%
f81fbe1113db6ca4c25ec54ed2e04f42	47.95%	105 of 219	-12.79%	-11.87%
f9b5afdff92f1eb5c870cf4b601e8dc1	3.89%	166 of 4269	-0.56%	-0.28%
fb891ea00a8758f573ce1b274f974634	20.68%	97 of 469	-0.21%	+0.00%
fbefbe3884f5a2aa209bfc96e614f115	41.95%	146 of 348	-16.09%	+7.47%
fd1af0690436028285a889c1928041ca	56.83%	79 of 139	-9.35%	0.00%
Average for 17x benign	38.49%		-8.38%	-2.45%
0018874837a567609e289661cd418639	17.10%	85 of 497	-4.30%	-0.60%
003d668ef73eef4aaa54a0deb90715de	22.39%	245 of 1094	-18.65%	-1.01%
12436ccaf406c2bf78cf6c419b027d82	39.77%	35 of 88	-11.05%	+2.27%
128629e7a3fd7f28ecff2039b5fd8b62	46.80%	476 of 1017	-30.07%	-12.78%
f181409e206cbe2a06066b79f1a39022	10.31%	234 of 2269	+1.06%	+1.94%
f3194dee0dc6e8c245dc94c5435750a5	13.17%	64 of 486	-1.64%	+0.82%
f342d8f0c18410e582441b19de8dd5bb	32.59%	305 of 936	-13.30%	-10.79%
f458ca5d41347a69c1c8dc99812485ee	10.05%	584 of 5813	-9.16%	-1.93%
f46f75e4eb294d5f92c0977c48f5af4f	15.83%	132 of 834	+18.35%	+19.18%
f4d80df6710b3848bf8c78c1b13fe3b5	14.81%	16 of 108	+9.87%	+25.93%
f55a7ad2ab8b3ac2447964614493fffe	14.15%	15 of 106	+10.21%	+26.42%
f7ad9e256725dd6c3cab06c1ab46fcc2	22.31%	620 of 2779	-11.71%	-7.16%
f98ae3c49ce8d4d5ec70f45f06601629	67.74%	21 of 31	-33.92%	+12.90%
fd225d8afd58cdec5f0c9b0f7fd77f58	41.34%	296 of 716	-19.07%	+2.23%
fd48609ba4ee42f05434de0a800929ad	52.00%	52 of 100	+9.76%	+9.00%
fdbce10ece29f14adfb7ebe99931d978	28.30%	30 of 106	+0.94%	+1.89%
fe3cb50833c74c60708e4e385bb8b4fc	8.74%	41 of 469	-6.25%	-1.28%
fead2a981fc24a2f9dd16629d43a6969	39.56%	36 of 91	-11.73%	+1.10%
Average for 18x malicious	27.61%		-6.70%	+3.79%

these apps heavily rely on WebKit capabilities and are in essence just an easy web browser where all the app's functionality is implemented on a server.

- TraceDroid performs slightly better than the Andrubis platform. This is caused by the fact that Andrubis uses a fixed timeout value (180 seconds in this case) which limits the number of different simulations Andrubis can initiate. A shorter runtime also means that the monkey exerciser has less time to generate its sequence of events. We will see that the monkey exerciser is responsible for most method invocations in Section 5.3.2.
- Differences between manual and automated Andrubis analysis are not excessive. The differences between manual analysis and TraceDroid are even smaller and gives us hope that we can improve Andrubis as well. It is obviously not expected that the currently used automated

simulations outreach manual analysis, due to the complicated nature of most applications.

Understanding low code coverage results

Table 5.6 shows us that code coverage is relatively low (< 40%), even for manual analysis. As it is desired to understand why this is the case, we analyzed the log output in more detail and conclude that there are a number of reasons that may have a negative effect on the code coverage numbers.

External libraries Many apps include external libraries that are used for a variety of purposes. It is unlikely that an app uses the complete feature set of an external library, which causes a lower percentage of code covered. Methods from these libraries that are not invoked by the app, have thus a negative effect on the percentage of code covered. If an app includes large libraries, it is likely that the coverage results drop significantly.

External libraries may be generalized into three classes: advertisement APIs; APIs for component access; and vendor-specific libraries. Most libraries seem to relate to the first two classes: processing advertisements (e.g., Google's AdMob[1] or AMoBee's Ad SDK[2]) and component access APIs (e.g., social media APIs for Twitter[3] or Facebook[4] or special JSON or XML parsers[5,6]). Vendor-specific libraries are found in apps developed using a visual development environment (such as MIT's App Inventor[7] or commercial software like AppsBuilder[8]), but may also be special helper libraries that appear in all apps developed by the same company. We are, unfortunately, not yet able to distinguish and exclude external libraries automatically other than by using a whitelist.

To illustrate the impact that external libraries have on the code coverage, we took a closer look at the sample with MD5 hash `12b7a4873a2adbd7d4b89eb17d57e3aa`. Table 5.6 shows that 2644 methods were missed during dynamic analysis. Analyzing the code coverage log output teaches us that of these missed calls, an immense 2522 methods are external library functions (Twitter: 1293, Facebook: 753, OAuth[9]: 160, Google (data, analytics, ads): 112, and vendor-specific: 204). Recomputing the code coverage while excluding these libraries resulted in an increase of the coverage percentage of more than 40%: from 7.55% to 49.61%.

Unreachable code As with normal x86 applications, apps are likely to contain a number of functions that are (almost) never executed. These include specific exception handlers or other methods that are only reachable via an improbable branch.

Complex applications When manually analyzing large complex applications such as games for only 180 seconds, it is likely that the analyzer does not have

[1] `http://www.google.com/ads/admob`
[2] `http://www.amobee.com/technology/ad_sdk.shtml`
[3] `http://twitter4j.org`
[4] `http://developers.facebook.com/android`
[5] `http://jackson.codehaus.org`
[6] `http://kxml.sourceforge.net`
[7] `http://appinventor.mit.edu`
[8] `http://www.apps-builder.com`
[9] `http://code.google.com/p/oauth-signpost`

enough time to complete all levels or to trigger all options and thus 'unlock' new method regions in the codebase. This is even harder when there is no knowledge about the app's semantics at all, as is the case during automated analysis. Thus the monkey exerciser is unable to simulate all the available options that are provided by the application.

5.3.2 Breakdown of simulation actions

To understand how code coverage is distributed among the different simulation actions and to determine which action is responsible for which percentage of code coverage, we analyzed the ANDRUBIS sample set while keeping track of the simulation intervals. To ensure a clean environment, each sample was reinstalled between two simulation actions. It must be noted that this approach limits the total percentage of code covered since receivers or timers installed during simulation x, will be lost during simulation $x + n$.

We ran analysis using both the ANDRUBIS platform and TRACEDROID. The following list describes the simulation groups as identified for ANDRUBIS.

Common Send text messages and initiate phone calls.
Broadcast Send intents to all broadcast receivers found in the manifest.
Activities Start all exported activities found in the manifest.
Services Send intents to all services found in the manifest.
Monkey Monkey exerciser.

The list of simulation actions applicable for TRACEDROID can be found in Table 4.2 on page 36. The breakdown results for ANDRUBIS are listed in Table 5.7, while Table 5.8 depictes the results for TRACEDROID. In these tables, **sum** is the total percentage of code that was covered during analysis. Due to overlapping, this does not equal the sum of the coverages during individual simulation rounds. It must also be noted that ANDRUBIS failed analysis on some samples. These failures will be discussed later in Section 5.4.

Table 5.7: ANDRUBIS breakdown

Set	Common	Broadcast	Activities	Services	Monkey	Sum
219x benign	0.00%	0.79%	21.83%	0.56%	24.81%	27.74%
210x malicious	0.00%	4.68%	15.14%	7.14%	19.17%	27.80%

Studying the results, we observe the following behavior.

- Activity simulation and monkey exercising (which also visits numerous activities) are responsible for the largest portions of code coverage. This is due to the fact that activities are, in general, main entry points for an application and often contain method invocations for initializing objects, installing action listeners, and setting viewpoints. It is expected for the monkey exerciser to gain the highest amount of code coverage as its randomized sequence of input events (pressing buttons, selecting options, switching tabs, etc.) can result in the execution of new application components.
- The ANDRUBIS **common** simulation group did not initiate any method invocations. Inspection of the framework teaches us that this is caused by

Table 5.8: TraceDroid breakdown

Set	B	i	o	I	O	N	L	P	M	A	S	E	sum
250x benign	0.43%	0.07%	0.00%	0.04%	0.01%	0.03%	0.00%	0.04%	20.00%	22.50%	0.69%	27.29%	29.93%
242x malicious	6.90%	1.63%	0.20%	1.26%	0.80%	0.52%	0.39%	0.04%	16.46%	20.23%	6.96%	26.06%	32.59%

`B`: boot
`i`: incoming text message
`o`: outgoing text message
`I`: incoming phone call
`O`: outgoing phone call
`N`: network disconnect
`L`: low battery
`P`: package removal/install/update
`M`: main activity
`A`: activities
`S`: services
`E`: monkey exerciser

its implementation: the operations listed under the **common** simulation round are non-blocking. This means that after the last emulated event, the app was immediately uninstalled and it was given no time to execute any method.

- Malicious applications tend to initiate more services than their benign counterparts. This comes not unexpected: services are allowed to run in the background and offer a malware writer possibilities to secretly send data to a remote server. On a similar note, we see that common phone activities such as receiving text messages or receiving phone calls are of limited interested for benign applications, while malicious apps are more attentive. SMS text messages, for example, could be used by a mobile botnet for C&C communication, while a banking trojan could forward detected mobile TAN (Transaction Authentication Number) codes to a remote server.
- From Table 5.8 we can confirm our hypothesis from Section 5.3.1 that code coverage for malicious applications is higher than for benign apps. This is indeed caused by the fact that a larger portion of malicious code is activated during the special simulation rounds (mainly boot simulation and text message receival).

5.3.3 Coverages results

Analysis was repeated without reinstalling the package between each simulation round. It was expected that this would have a positive effect on the coverage results, as receivers or services started during round x may now be activated in round $x + n$. Results are depicted in Table 5.9 while a cumulative distribution function (CDF) is shown in Figure 5.2.

Aside from the code coverage results, Table 5.9 also includes the percentage of detected uncaught exceptions and VM crashes. This first number indicates the percentage of applications that threw an unexpected exception during analysis (mostly a `NullPointerException` or the more general `RuntimeException`) and points to faulty apps as such exceptions should — under normal circumstances — always be caught. The second number illustrates the percentage of

Table 5.9: Code coverage results

Platform	Set	Code coverage	Uncaught exceptions	VM crashes
ANDRUBIS	233x benign	26.76%	18.88%	13.73%
	231x malicious	27.29%	49.13%	6.09%
TRACEDROID	250x benign	31.10%	24.00%	3.20%
	242x malicious	35.02%	45.87%	1.65%

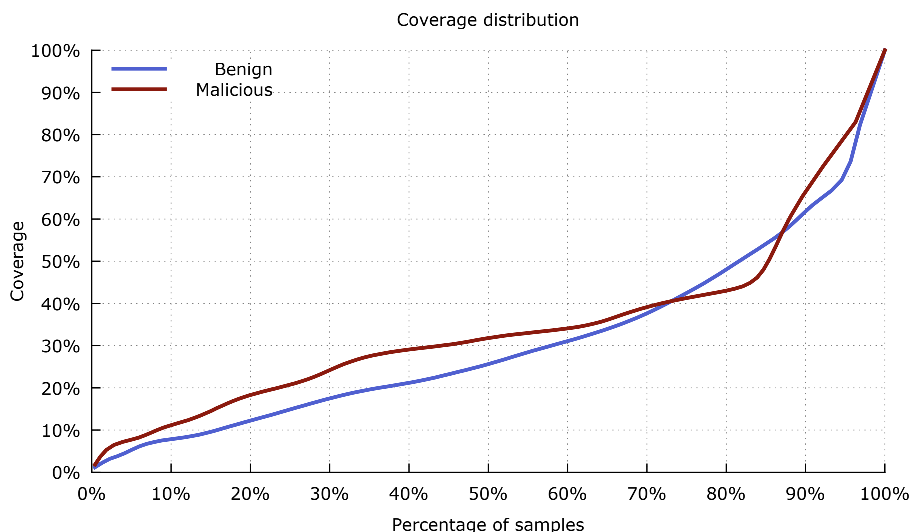


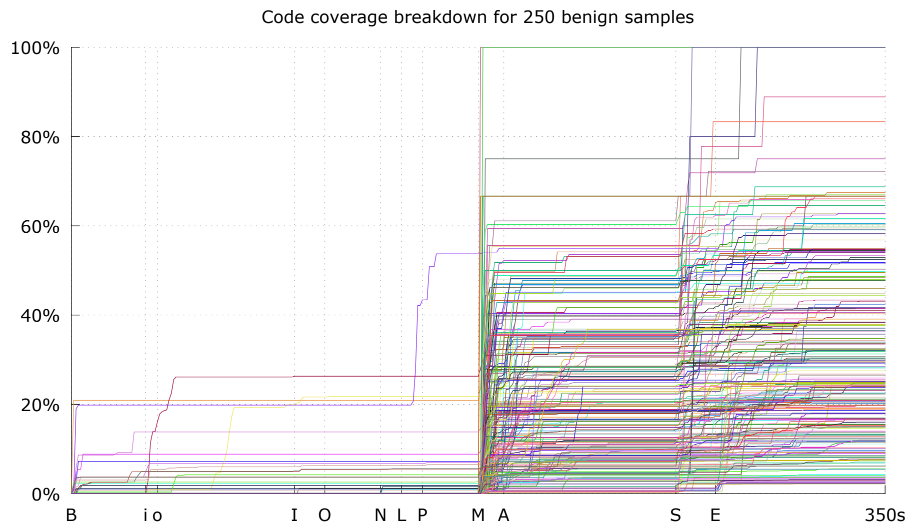
Figure 5.2: CDF for TRACEDROID coverage results

apps that caused a complete VM crash during analysis. This normally indicates a bug in the native code of the VM and may be related to the TRACEDROID implementation.

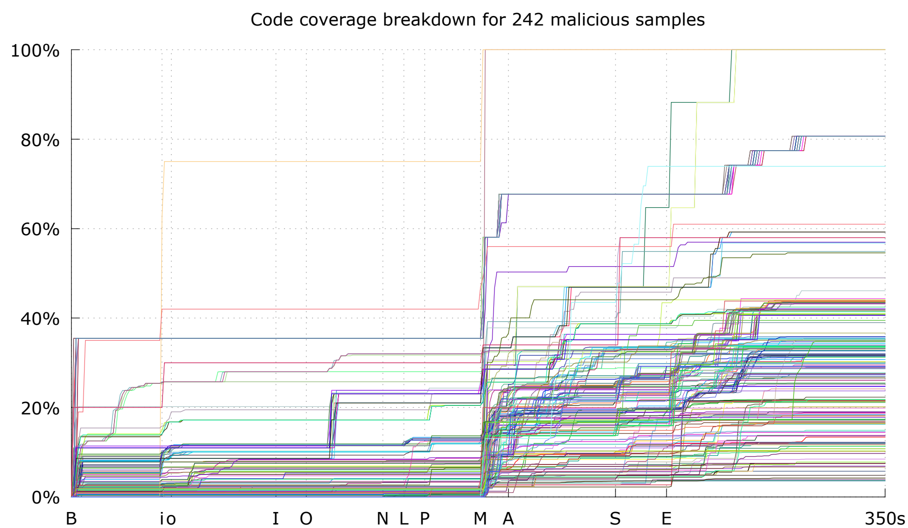
Surprisingly, the code coverage results for ANDRUBIS from Table 5.9 are worse than when we computed the coverage for each simulation separately in Section 5.3.2. We feel that this is probably caused by the true randomness of the ANDRUBIS monkey exerciser setup as discussed earlier. Especially since we do see an increase in code coverage percentage for TAP.

Furthermore, we identify differences between the percentage of uncaught exceptions. Studying the results in more detail tells us that this is caused by ANDRUBIS not being able to analyze some of the samples. We also see that the number of VM crashes for TAP are lower than for ANDRUBIS. These failures are outlined in more detail in Section 5.4.

From Figure 5.2, we deduce that for 80% of the samples a code coverage of 50% or less was achieved. This corresponds to our earlier observation that, on average, code coverage is relatively low. Also noticeable is the drop around 75–85% for malicious samples compared to the benign set. After studying the results in more detail, we conclude that this dip is caused by a cluster of malware samples that are likely related to each other (i.e., they are from the same malware family).



(a) Benign



(b) Malicious

Figure 5.3: Code coverage breakdown per simulation

Overall, we conclude that both analysis platforms are able to gain an average code coverage of about 30%, which, compared to manual analysis results discussed earlier, is a decent value and should provide a good insight in the app's capabilities. TAP performs slightly better than the Andrubis platform, which is likely caused by the more extensive set of simulation events and the lack of a fixed runtime window.

We conclude with two figures that illustrate the increase of code coverage per second for all analyzed samples in Figure 5.3. The x-tics are set on the start of a new simulation action and are named according the abbreviations used in

Table 5.8.

The plots from Figure 5.3 confirm our previous statements and expectations that malicious apps are attentive for the special simulations like reboot emulation (so that background services can be started as soon as a possible) and emulation of an incoming text message.

5.4 Failures

A small variety of failures were detected during the different number of automated analysis runs executed on our sample set. To make statements on the impact of these failures on the analysis frameworks, we took a closer look at those detected during the breakdown analysis runs from Section 5.3.2.

Failures were detected by inspecting the code coverage output files. The coverage post processing script already detects and logs VM crashes by searching for dumped stack traces in the logcat output files. We grouped these crashes in an **acceptable** subclass, as they do not have a direct effect on the overall functioning of the frameworks. In the case of non-existent coverage log files, however, we identified a **severe** error (timeout) as this indicates that the automated analysis failed to complete successfully.

The output directories of the failed samples were examined in more detail to classify failures into different sub-classes. The classification of failures detected during the breakdown benchmark are depicted in Table 5.10.

Table 5.10: Classification of detected failures

		Severe				Acceptable				
Platform	**Set**	**Time[1]**	**Time[2]**	**Time[3]**	**Total**	**Webcore**	**Unknown**	**Sigstkflt**	**Setuid**	**App**
ANDRUBIS	250x benign	15	3	13	31 (12.4%)	5	6	24	0	0
	242x malicious	15	9	8	32 (13.2%)	1	5	20	0	0
TRACEDROID	250x benign	0	0	0	0 (0%)	8	0	0	0	0
	242x malicious	0	0	0	0 (0%)	1	0	0	3	1

Besides the dynamic analysis runtime window of 240 seconds, ANDRUBIS also installs a watchdog alarm that kills ongoing analysis if it does not finish in time. This value defaults to 4 times the runtime window, but was increased to 8 times the runtime value since the package was reinstalled between each simulation round, introducing a possible extra overhead. We ran into a couple of samples that still did not manage to finish in time. We distinguish three different classes of these severe timeout issues.

Time[1] Analysis was interrupted during the simulation of an event. Investigation showed that the timeouts were caused by a very long delay during the app's installation. The reason for this is likely the modified DROIDBOX setup that floods the logcat with output data obtained during the installation of a package.

Time[2] All simulation rounds were completed successfully but no code coverage percentage was computed. This happens when an application invokes a lot of methods, causing a slow down in the code coverage script which ultimately triggers the global watchdog alarm.

Time3 Due to an error in the analysis framework, previous emulator instances were sometimes not killed correctly, causing the boot of a fresh emulator to fail. The analysis script then got stuck in an endless loop until it was killed by the watchdog alarm.

The details for **acceptable** failures are more interesting and require some more debugging as they trigger a crash of the Dalvik VM during execution.

Webcore Some samples crash with a `jarray points to non-array object` warning, followed by a stack trace of the VM. A Google search revealed that this is a known bug in the emulator sources for Android 2.3.4[10]. Unfortunately, no other fix is available but to upgrade to a newer platform release.

Unknown This is a reproducible bug that occurs only on the ANDRUBIS platform and only when method tracing is enabled. The bug does not occur on TAP with method tracing in place. It originates from the `objectToString()` function and is likely caused by the combination of TAINTDROID and TRACEDROID and requires further investigation by the ANDRUBIS source code maintainers. A short debugging session using the `arm-eabi-addr2line` tool (Listing 5.1) illustrates that this is a complicated bug as it gets triggered during TRACEDROID's `.toString()` invocation for object resolution. Depending on the object's `.toString()` implementation, numerous methods may have been invoked before the error comes into view. This makes it hard to determine the exact method that is responsible for the damage.

Sigstkflt The stack fault signal `SIGSTKFLT` is raised by the Android OS to trigger a debugger dump when an app becomes unresponsive. Again, as with the **unknown** bug, these signals were only detected on the ANDRUBIS platform. Inspection of the stack trace leads to a race condition in the `LOGD_TRACE()` function which causes a deadlock. Fortunately, these failures occur only after method tracing was stopped using the `am profile stop` command and thus have very limited effect on the analysis results. The fact that the bug cannot be reproduced within TAP makes it harder to investigate in more detail.

Setuid Some samples printed a `cannot setuid(10033): Try again` warning message before their VM crashed. Investigation of the crash reveals that this is not an implementation issue, but the *Rage Against The Cage* exploit[11], spawned by a malicious application, that is trying to obtain root priviliges. The vulnerability exploited by *Rage Against The Cage* was patched since Android 2.3 and does thus not function within TRACEDROID and ANDRUBIS.

App One sample crashed due to an error in one of the app's native libraries.

Two other bugs were detected during analysis. The first relates to the way Python decodes zip headers[12] and was easily fixed by patching `zipfile.py`. The second issue relates to a bug in the Android qemu drivers for emulating a

[10] http://code.google.com/p/android/issues/detail?id=12987
[11] http://dtors.org/2010/08/25/reversing-latest-exploid-release
[12] http://bugs.python.org/issue14315

Listing 5.1: Stack trace with added method resolution for the **unknown** bug

```
pid: 536, tid: 536  >>> com.putitonline.da <<<
signal 11 (SIGSEGV), code 1 (SEGV_MAPERR), fault addr 0000007c
 r0 00000000  r1 00000000  r2 00000034  r3 00000000
 r4 be8bfda8  r5 4850f750  r6 0000ce60  r7 be8bfda8
 r8 00000030  r9 be8bfdf0  10 48519450  fp 4850f768
 ip aca75cb9  sp be8bf8d8  lr aca681c3  pc aca6f436  cpsr 40000030
         #00  pc 0006f436  libdvm.so dvmSlotToMethod()    in Reflect.c
         #01  pc 0006813e  libdvm.so
     Dalvik_java_lang_reflect_Method_getMethodModifiers() in java_lang_reflect_Method.c
         #02  pc 00027144  libdvm.so dvmInterpretDbg()    in InterpC-portdbg.c
         #03  pc 0001c5a0  libdvm.so dvmInterpret()       in Interp.c
         #04  pc 000603ec  libdvm.so dvmCallMethodV()     in Stack.c
         #05  pc 000606e2  libdvm.so dvmCallMethod()      in Stack.c
         #06  pc 0004f5cc  libdvm.so objectToString()     in Profile.c
         #07  pc 0004f788  libdvm.so parameterToString()  in Profile.c
         #08  pc 0004f850  libdvm.so getParameters()      in Profile.c
         #09  pc 0004fa7e  libdvm.so handle_method()      in Profile.c
         #10  pc 0004fc22  libdvm.so dvmMethodTraceAdd()  in Profile.c
         #11  pc 00031de8  libdvm.so ...
         ...
```

GPS fix[13]. While a patch is available[14], it was decided to temporarily disable GPS simulation during analysis.

During analysis of the Andrubis data set using the TraceDroid platform, no failures were found that relate to the VM modifications described in Chapter 4. The webcore bug was detected for 9 samples (1.83%), but is unrelated to our modifications. The other detected 'failures' (three times **setuid** (0.61%) and once an **app**'s own native code (0.20%)) are solely app based.

Slightly more failures were detected during the Andrubis analysis run. For 63 samples (12.80%) analysis failed to complete in time. These timeouts, however, are unrelated to our VM modifications. The 44 **sigstkflt** and 11 **unknown** errors (8.94% and 2.24% respectively) do relate to the method tracing implementation, but only the latter set has an impact on the results. Since these errors were not seen during the TraceDroid analysis runs, they must be caused by a combination of TaintDroid and the new method tracer. The number of **unknown** failures is so low, however, that we can conclude that our Dalvik VM modifications outlined in Chapter 4 prove to be stable enough and are suitable for use in a production environment for automated analysis.

5.5 Dissecting Malware

In this section, we analyze a malicious Android application to demonstrate our framework's capabilities. After generating a clustered call graph and inspecting the extracted features, we quickly identify suspicious code executions to narrow our further analysis. We show how our approach eliminates the need for deobfuscation and demonstrate the profit gained by running manual dynamic analysis to stimulate broadcast receivers in a more fine-grained matter.

[13] http://code.google.com/p/android/issues/detail?id=13015

[14] http://android.googlesource.com/platform/sdk/+/35425faccd6c6591c787f69dfb8e845720ca15ac^!

5.5.1 ZitMo: ZeuS in the Mobile

Our example is a mobile variant of the ZeuS trojan horse family. ZeuS' botnets are estimated to include millions of compromised computers and are used to collect personal information of victims that include credentials for social networks or online bank accounts [34]. For the latter, PC-based ZeuS uses a scheme wherein the bank's official webpage is modified so that money can be transferred to arbitrary accounts.

To prevent these attacks, banking services introduced the use of mobile Transaction Authentication Number (TAN) messages as a form of two-factor authentication. When a transaction is initiated, a TAN is generated by the bank and sent to the user's mobile phone by SMS. To complete the online transaction, the user has to insert the received TAN into the bank's webpage. The received SMS message may contain additional information about the transaction such as account number and amount of money that will be transferred.

The mobile ZeuS variant is used in addition to PC-based ZeuS to complete malicious transactions. By intercepting and forwarding mTAN messages to a remote server, it bypasses the two-factor authentication scheme [46]. PCs infected with ZeuS trick users in installing the malicious app by stating that their phone needs be activated as part of extra security measurements. Once the victim entered his phone number, a text message is sent to the phone that contains a link to the malicious application.

5.5.2 Dissecting `a1593777ac80b828d2d520d24809829d`

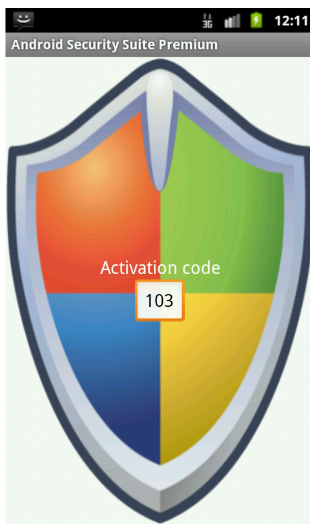
We ran our dynamic analysis tool on the ZitMo malware sample with MD5 hash `a1593777ac80b828d2d520d24809829d` of which VirusTotal reports that it was detected as malicious by 32 out of 46 anti-virus vendors[15]. After completion of the automated analysis run, we first have a quick look at the generated screenshot during analysis of the main activity as depicted in Figure 5.4a.

The screenshot shows a huge security logo that contains an activation code. It appears that there are very little possibilities to interact with the app. This is confirmed by inspecting the output of the code coverage processing script while using the `--interval` option. The graph for the coverage distribution as depicted in Figure 5.4b clearly shows that the monkey exerciser has very limited effect on the overall percentage of code coverage.

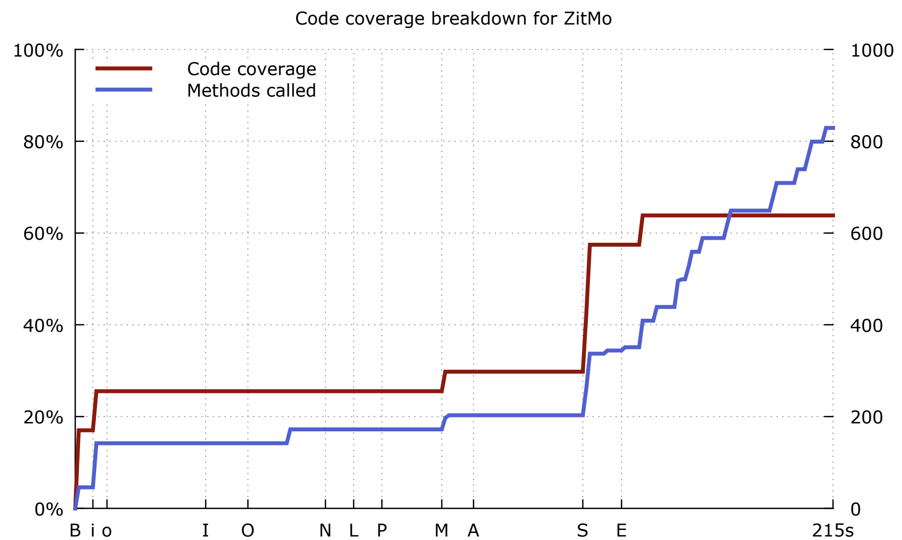
To get an overview of the app's internal data flow, we generated a call graph that is depicted in Figure 5.5 on page 67. Colored, clustered output was used to easily identify the origin of method invocations between the different components of the application. API calls were omitted to reduce the size of the graph.

Studying the call graph, we observe that this particular ZitMo variant does not obfuscate its method or class names which eases analysis. We identify a receiver named `SecurityReceiver` with a suspicious `GetLastSMS()` method. Noteworthy is also the `MakeHttpRequest()` method that is responsible for making a HTTP request via Apache's `HttpURLConnection` class. If we recall that the app has a very limited set of possible interactions, this indicates that the HTTP request is likely initiated as a reaction to one of our simulated events.

[15] `https://www.virustotal.com/en/file/8ae9e08578b24ad61385eebbc17d78b0230e9177/analysis`



(a) Main activity



(b) Code coverage distribution

Figure 5.4: ZitMo

Our next step involves constructing a list of features that may indicate malicious behavior. Listing 5.2 displays how we load the output log directory and generate the feature set. Its outcome confirms that there was some network activity initiated by the app during the analysis session. It also shows that some personal data was read by the application that includes the phone's Internal Mobile Station Equipment Identity (IMEI) and International Mobile Subscriber Identity (IMSI), as well SMS reading or writing activity.

Listing 5.2: Generating a feature set for ZitMo

```
./trace --logdir a1593777ac80b828d2d520d24809829d.2013-07-14.14.08.57.143089
Dropping an ipython shell. You can now play with the traces.
In [1]: import features
In [2]: f = features.Features()
In [3]: f.get_features(traces, api_classes, 'unknown')
In [4]: f.dump()
...
io_database      : True
network          : True
telephony_imei   : True
telephony_imsi   : True
telephony_msisdn: True
telephony_sms    : True
```

We start with dissecting the HTTP request. Continuing the current `trace` session, in Listing 5.3 we search for invocations of `getResponseCode()`.

The sample forwards our received message to a remote webpage at `http://android2update.com/biwdr.php`. To understand how the request URL is constructed, we search for method invocations that return the URL's parameters. This is illustrated in Listing 5.4.

Listing 5.3: `getResponseCode()` invocation

```
In [5]: for f in functions:
   ...:   if f.name == 'getResponseCode': print f.target_object_s
'org.apache.harmony.luni.internal.net.www.protocol.http.HttpURLConnectionImpl:
   http://android2update.com/biwdr.php?to=15555215403
                &i=310260000000000&m=000000000000000&aid=103&h=0&v=1.2.3
                &from=4224&text=incoming+text+message+XLastMessage&last=1'
```

Listing 5.4: Retrieving URL parameters

```
In [6]: for f in functions:
   ...:   if f.return_value == '310260000000000':
   ...:     print '%s.%s()' % (f.target_object, f.name)
android.telephony.TelephonyManager.getSubscriberId() #IMSI

In [7]: for f in functions:
   ...:   if f.return_value == '000000000000000':
   ...:     print '%s.%s()' % (f.target_object, f.name)
android.telephony.TelephonyManager.getDeviceId()     # IMEI

In [8]: for f in functions:
   ...:   if f.return_value == '103':
   ...:     print '%s.%s()' % (f.target_object, f.name)
com.android.security.ValueProvider.GetActivationCode()
```

It is also interesting to see if there is a special method that dynamically constructs the base URL in order to hinder static analysis. In Listing 5.5, we first search for functions that return the base URL, followed by printing the method traces for this particular function. Note that some output was omitted or reformatted to maintain readability.

Listing 5.5: Domain name deobfuscation

```
In [9]: for f in functions:
   ...:   if f.return_value == 'http://android2update.com/biwdr.php':
   ...:     print '%s.%s()' % (f.target_object, f.name)
com.android.security.ValueProvider.GetAntivirusLink()

In [10]: for f in functions + constructors:
   ....:   if isinstance(f.called_by, Function) and
                          f.called_by.name == 'GetAntivirusLink':
   ....:     print '%s(%s).%s(%s);' % (f.target_object,f.target_object_s,f.name,
                                        f.parameters)
   ....:     print 'return "%s"'    %  f.return_value
java.lang.String("qh't,;t>p%;%:%>/q/a<qndq%roi>qdq2up,d%a>=tqe.cqo,%m/,bi-w=dr.p,h'p").
                                                               replace( '[',  '');
          return "qh't,;t>p%;%:%>/q/a<qndq%roi>qdq2up,d%a>=tqe.cqo,%m/,bi-w=dr.p,h'p"
java.lang.String("qh't,;t>p%;%:%>/q/a<qndq%roi>qdq2up,d%a>=tqe.cqo,%m/,bi-w=dr.p,h'p").
                                                               replace( '=',  '');
          return "qh't,;t>p%;%:%>/q/a<qndq%roi>qdq2up,d%a>tqe.cqo,%m/,bi-wdr.p,h'p"
# output omitted for readability
java.lang.String("http%%:%//and%roid2upd%ate.co%m/biwdr.php").replace( '%',  '');
          return "http://android2update.com/biwdr.php"
```

We see that the base URL is first read as a very obfuscated string and gradually gets deobfuscated by calls to `String.replace()` to remove superfluous

characters. A similar approach can be used to print the method trace for a specific function. In our process of disassembling the internals of `SecurityReceiver`, consider Listing 5.6 for printing the method trace of the suspicious `GetLastSms()` method.

Listing 5.6: Method trace for `GetLastSms()`

```
In [11]: for f in functions:
   ....:   if f.name == 'GetLastSms':
   ....:     print '%05d - %05d' % (f.linenumber_enter, f.linenumber_leave)
00489 - 00536
In [12]: for f in functions + constructors:
   ....:   if f.linenumber_enter >= 489 and f.linenumber_leave <= 536:
   ....:     if isinstance(f, Function):
   ....:       print '%s %s %s.%s()' % (' '*f.depth,f.return_type,f.target_object,f.name)
   ....:       print '%s return %s'  % (' '*f.depth,f.return_value)
   ....:     if isinstance(f, Constructor):
   ....:       print '%s new %s()'   % (' '*f.depth,f.class_name)
# output reformatted for readability
com.android.security.NumMessage com.android.security.SecurityReceiver.GetLastSms()
 android.net.Uri android.net.Uri.parse()
 return 'content://sms/inbox'
 android.content.ContentResolver android.content.ContextWrapper.getContentResolver()
 return 'android.app.ContextImpl$ApplicationContentResolver@4053e060'
 android.database.Cursor android.content.ContentResolver.query()
 return 'android.content.ContentResolver$CursorWrapperInner@40537e70'
 boolean android.database.CursorWrapper.moveToFirst()
 return true
 int android.database.CursorWrapper.getColumnIndexOrThrow()
 return 11
 java.lang.String android.database.CursorWrapper.getString()
 return 'incoming text message'
 int android.database.CursorWrapper.getColumnIndexOrThrow()
 return 2
 java.lang.String android.database.CursorWrapper.getString()
 return '4224'
 java.lang.StringBuilder java.lang.StringBuilder.append()
 return 'incoming text message'
 java.lang.StringBuilder java.lang.StringBuilder.append()
 return 'incoming text message XLastMessage'
 new com.android.security.NumMessage()
return com.android.security.NumMessage@4053ba70
```

As its name already reveals, we see that the `GetLastSms()` method fetches the latest received SMS text message from the user's inbox and returns it as a `NumMessage` object.

Continuing our study, we open the dumped method trace file and search for more interesting execution traces. Depicted in Listing 5.7, We find a number of string comparisons called by the `AlternativeControl()` method that deserve some more attention.

Listing 5.7: Method trace for `AlternativeControl()`

```
public boolean java.lang.String("incoming text message").startsWith("%"); return false
public boolean java.lang.String("incoming text message").startsWith(":"); return false
public boolean java.lang.String("incoming text message").startsWith("*"); return false
public boolean java.lang.String("incoming text message").startsWith("."); return false
```

The `AlternativeControl()` method seems to test whether the first character of the emulated text message matches a particular predefined character. In Listing 5.8, we initiate another analysis session and restart analyzing the sample. This time, however, we provide the `--manual` flag in order to have full control over the content of emulated SMS messages.

Listing 5.8: Manual dynamic analysis

```
./analyze.py --input ../apks/zitmo/a1593777ac80b828d2d520d24809829d --manual
...
In [1]: self.emu.sms_recv(1234,'%44444444')
In [2]: self.emu.sms_recv(1234,':33333333')
In [3]: self.emu.sms_recv(1234,'*22222222')
In [4]: self.emu.sms_recv(1234,'.11111111')
```

We can now use the new method trace to reconstruct the control flow of `AlternativeControl()`. Received SMS messages that start with a `%` sign indicate an info request. `AlternativeControl()` will send an SMS text message containing device information to a phone number that is extracted from the incoming message. Once finished, the SMS received broadcast is aborted so that it will not appear in the user's inbox. This is depicted in Listing 5.9.

Listing 5.9: Method trace for `AlternativeControl()`

```
public boolean com.android.security.SecurityReceiver().AlternativeControl("%44444444")
 public boolean java.lang.String("%44444444").startsWith((java.lang.String) "%")
 return (boolean) "true"
 public java.lang.String
 com.android.security.SecurityReceiver().ExtractNumberFromMessage("%44444444")
 return (java.lang.String) "+44444444"
 public void
 com.android.security.SecurityReceiver().SendControlInformation("+44444444")
  public static boolean com.android.security.ValueProvider.IsTotalHideOn()
  return (boolean) "false"
  public static boolean com.android.security.ValueProvider.IsAlternativeControlOn()
  return (boolean) "false"
  public static java.lang.String com.android.security.ValueProvider.GetActivationCode()
  return (java.lang.String) "103"
  public static java.lang.String java.lang.String.format((java.lang.String)
      "Model:%s AC:%s H:%d AltC:%d V:%s Mf:%s/%s", [Ljava.lang.Object;@40533340)
  return (java.lang.String) "Model:generic AC:103 H:0 AltC:0 V:1.2.3 Mf:unknown/2.3.4"
  public static void com.android.security.SecurityReceiver.sendSMS("+44444444",
      "Model:generic AC:103 H:0 AltC:0 V:1.2.3 Mf:unknown/2.3.4")
   public static android.telephony.SmsManager android.telephony.SmsManager.getDefault()
   return (android.telephony.SmsManager) "android.telephony.SmsManager@40534448"
   public void android.telephony.SmsManager("android.telephony.SmsManager@40534448").
      sendTextMessage("+44444444", "null",
                      "Model:generic AC:103 H:0 AltC:0 V:1.2.3 Mf:unknown/2.3.4",
                      "null", "null")
   return (void)
  return (void)
 return (void)
return (boolean) "true"
final public void android.content.BroadcastReceiver().abortBroadcast()
return (void)
```

Examining the trace output file in more detail, we can determine the purpose

of `AlternativeControl()`. For this particular malware sample, alternative control stands for the use of SMS text messaging instead of Internet connectivity to distribute personal information. Alternative control can be enabled by sending a `:<phone-number>` message to the infected phone. Once enabled, all incoming text messages will be forwarded via SMS to the specified phone number. It can be disabled again by sending a text message that starts with a dot (`.`). Finally, a message starting with `*` seems to disable the software entirely.

5.5.3 Discussion

In this section, we illustrated the power of our analysis framework by analyzing an existing malicious application. We successfully identified and reconstructed core components of the app while using only dynamic analysis output results. Combined with existing static analysis tools, we believe that the framework can greatly improve the speed of which unknown applications are disassembled and their internal, potentially malicious, operation is revealed.

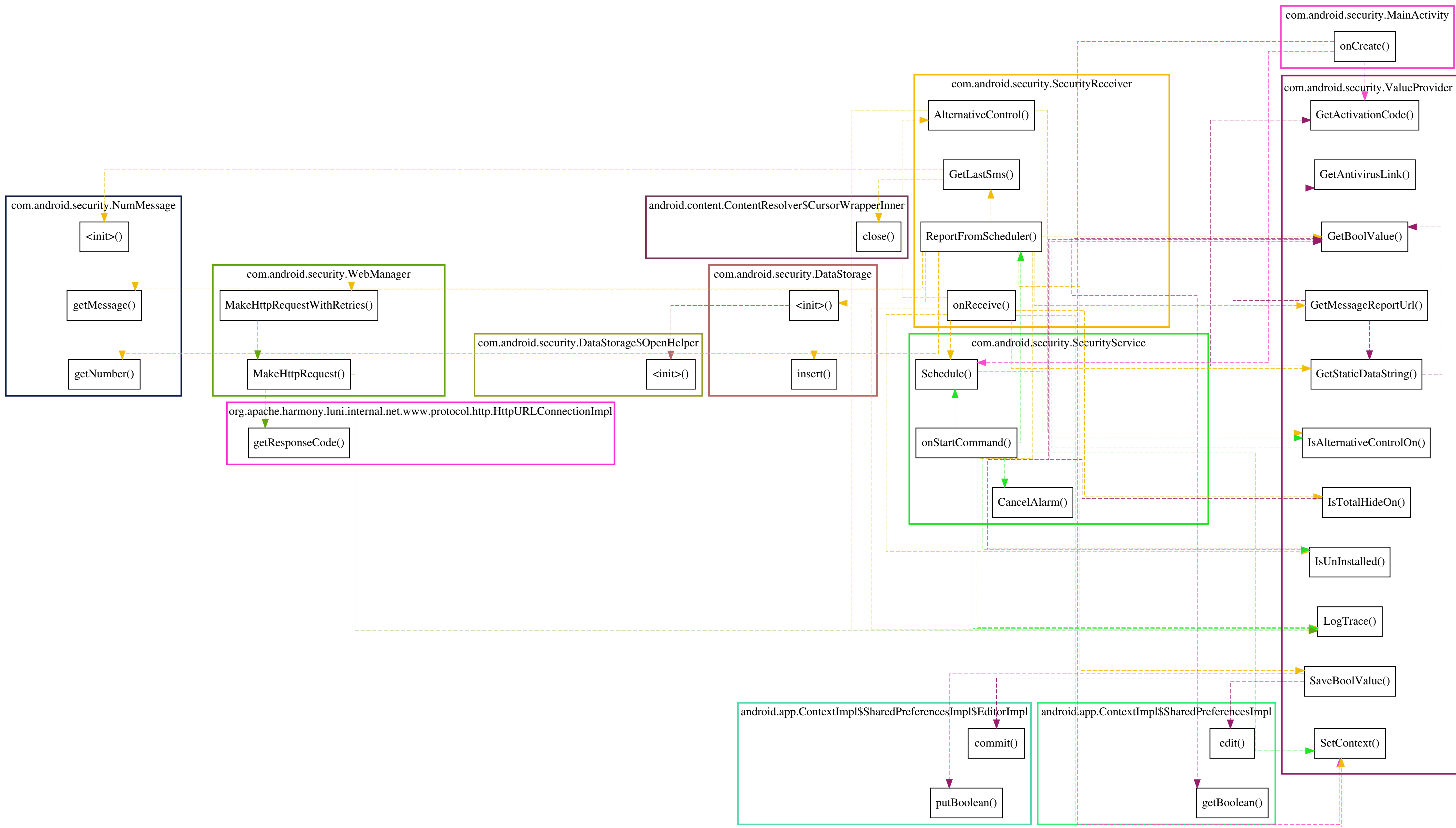


Figure 5.5: Call graph for ZitMo

Chapter 6

Related Work

Research focusing on Android security and Mobile malware in general has been an increasingly popular topic over the last decade. In this chapter, we provide an extensive overview of related work in the field of Android malware.

We first outline background and survey studies in Section 6.1. Next, in Section 6.2, we provide a systematization of knowledge regarding a broad range of Android research proposals. We conclude with a more detailed comparison between existing dynamic analysis platforms and our TraceDroid implementation in Section 6.3.

6.1 Background and Surveys

A number of studies focus on analyzing Android's security mechanisms. Before the first Android phones were even released in 2008, Schmidt et al. were one of the first to discuss the security of Android smartphones with a focus on its Linux side [61]. They state that Android's open character represents a great opportunity for researching security aspects on mobile devices. One of the first studies done on Android's permission model was done by Enck et al. [25]. This work details Android's internal components and their interaction.

In 2010, Shabtai et al. performed a comprehensive security assessment on the Android framework [62]. They list a number of possible countermeasures for tackling indicated high-risk threats to Android. After three years of research and Android development, most of these threats are still applicable:

1. Maliciously using the permissions granted to an installed application.
2. Exploiting a vulnerability in the Linux kernel or system libraries.
3. Exposing private content.
4. Draining resources.
5. Compromising the internal/protected network.

The paper proposes several recommendations to improve Android's security mechanisms in respect to these threats. Most research started from that moment on focused on threat groups 1) and 3) as it turned out that a large part of Android malware originates from these threats. This is confirmed by a Symantec Research white paper from 2011 that addresses the motivations of recent

Android Malware [17]. This paper concludes that the current mobile monetization schemes has a low revenue-per-infection ratio. It was expected that this ratio would increase when more devices store credentials backed by monetary funds. Something realized less than a year in later already in 2012 with the *Eurograbber* attack campaign, responsible for stealing over €36,000,000 [39].

In 2011, Enck discusses the current state of smartphone research and offers directions for future research [23]. In the same year, Enck et al. study 1100 popular free Android applications using the `ded` decompiler.

A survey paper from 2011 by Becher et al. provides an overview of mobile network security and used attack vectors and make statements on future research [5]. Similar work was done by Vidas et al. [67].

A first analysis on actual mobile malware was done by Felt et al. and studied 46 pieces of iOS, Android, and Symbian malware that spread in the wild from 2009 to 2011 [29]. A more extensive research done by Zhou and Jiang in 2012 covers 1260 Android malware samples distributed among 49 different malware families [78]. The huge data set was released to the research community as the *Malgenome Project*[1] and has ever since been used by many subsequent research papers. They categorize existing ways Android malware is distributed into three social engineering-based techniques: 1) repackaging; 2) update attacks; and 3) drive-by downloads. For all techniques described in the paper, however, users are always tricked in installing the — often over-privileged — malicious application them self. To the best of our knowledge, no known malware has the possibility to be installed on a user's device without his or her explicit consent.

6.2 Systematization of Knowledge

Ever since the first Android phones were released in 2008, researchers have proposed dozens of frameworks with a variety of purposes. In this section, we outline our efforts in systematizing these proposals. We enumerate this existing knowledge by means of an extensive reference table. For each proposal, we distinguish a number of characteristics and provide a brief summary. Based on their main purpose and approach, we classify the different research efforts into distinct categories.

In order to have a complete overview of all available Android related tools regarding Android's security mechanisms, we also take a number of open source applications into consideration. Although these tools are not necessarily results of ongoing research in the field of Android security, they may still be valuable for Android security researchers.

In the remainder of this section, we first describe the attributes that we use to characterize research efforts in Section 6.2.1. In Section 6.2.2, we describe the different categories a proposal can belong to. The final systematization of knowledge table can be found in Section 6.2.3.

6.2.1 Attributes

We group distinct attributes in a small number of subcategories: **type**, **technique**, and **deployment**. We outline these categories in more detail in the following paragraphs.

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

Type The type of a framework describes the kind of research that is addressed. We distinguish the following different types of frameworks:

Attack Proposals that address previously unseen *attack vectors* against the Android OS or its end-users.

Defense Proposals that describe novel *defense mechanisms* to protect end-users from specific malicious activity.

Analysis Proposals that describe the implementation or evaluation of an *analysis platform*. Such platform can be used by malware researchers or reverse engineers to gain information about an Android app's internals. TraceDroid belongs to this category.

Detector Proposals that address a scheme to automatically *detect* and report *malicious behavior*. These research topics may be of interest for anti-virus vendors as they could help increasing the detection rate of unknown malware.

Technique A proposed work may use a number of **techniques** indicating how the work is implemented. The following characteristics make it easy to group frameworks that use similar mechanisms:

Dynamic Frameworks that use a form of *dynamic analysis* to analyze unknown applications at runtime. TraceDroid belongs to this category as it has the capability to run targeted applications in a controlled environment to generate a behavioral footprint. Frameworks that protect users at runtime against specific malicious behavior also belong to this feature group.

VMI Dynamic analysis frameworks that use *Virtual Machine Introspection* (VMI) to intercept events that occur within the emulated environment [32]. VMI based systems are implemented on the emulator level (which is qemu in case of Android) and, as a consequence, are always prone to emulator evasion [55]. An advantage of VMI is that no OS modifications are required, which makes the analysis system highly portable.

Syscalls Dynamic analysis frameworks that collect an overview of executed *system calls*, by using, for instance, VMI or `strace`.

Method tracing Dynamic analysis frameworks that trace particular method invocations. TraceDroid is an example framework that uses method tracing.

Taint tracking Proposals that use a — possibly dynamic — *taint tracking* mechanism to protect a user's private data or to prevent memory corruptions within the kernel from happening.

Static Proposals that perform some kind of *static analysis* on targeted applications to obtain a footprint. These results may be used during a later processing stage of the framework's implementation or may be presented to the user directly. TraceDroid uses static analysis to efficiently stimulate a targeted application during the dynamic analysis stage.

Decompiler Proposals that implement a *Dalvik bytecode decompiler* or dissassembler.

Weaving Frameworks that rewrite bytecode of existing applications using a *bytecode weaving* technique similar to the one outlined earlier in Section 4.4.

Deployment If the proposed framework is available for download, the deployment group characterizes how it is deployed and implemented. For frameworks that are not available for download, we use the proposal's techniques to make

an estimated guess on how it is implemented. We distinguish the following deployment types:

Android app Proposals that are deployed as an Android application which could be installed by any Android user.

Android OS Proposals that are deployed as a modified Android OS to implement additional features.

Application Proposals that are deployed as an ordinary (web) application or script to perform some kind of analysis.

Proposals may use multiple deployment techniques. TraceDroid, for example, has both an *Android OS* and *application* component: the extensive method tracer is implemented as a modified OS, while a set of Python scripts are used to automate analysis and communicate with the platform.

Systems that are purely VMI based, are listed as sole applications as they only use a modified emulator while no OS changes are necessary.

Availability In addition to above characteristics, we also outline the availability of the discussed research by trying to obtain a working implementation copy of each discussed framework. The result of this search is reflected as *open* or *closed* source, commercially available (denoted as *paid*), remotely usable via a specifically designed *web* application or not available (denoted as *NA*). An overview of available frameworks and their accompanying URL is listed in Appendix B.

6.2.2 Classification

To structure our study of over 60 proposed frameworks, we classified them into 7 different categories. In the next paragraphs, we provide a brief description of the distinct classes.

1. Static Analysis This class describes analysis tools that perform static analysis on targeted Android applications. Most work categorized in this class is not a result from scientific research efforts, but are 'simple' tools to aid research analysts. These include decompilers and disassemblers like JEB and smali, but also analysis platforms that help researchers identifying malicious code like Androguard and Dexter.

2. Dynamic Analysis This class describes analysis platforms that run targeted applications in a controlled environment to obtain some kind of behavioral footprint. Results are normally presented to a user in the form of a report, possibly including a maliciousness rating. Most of these systems likely use some form of additional static analysis during an earlier stage to successfully perform the dynamic analysis. We discuss dynamic analysis platforms in more detail and compare them to TraceDroid in Section 6.3.

3. Dynamic Defenses This class describes defense mechanisms that protect users against malicious applications using a dynamic analysis approach. Most of these proposals are implemented as a modified Android OS to keep track of specific changes at runtime and a large number of these are specialized in

keeping track of some form of permission policy to prevent applications from accessing restricted components.

4. Static Defenses This class describes proposals that use static analysis to detect malicious or suspicious applications.

5. Attacks This class describes a small number of previously unseen attack vectors against the Android platform.

6. Repackaging This class describes a small number of proposals that detect or prevent the repackaging of Android apps. Repackaged apps are popular, legitimate applications that have been disassembled and repacked with an additional malicious payload. Users are tricked in installing such apps as the promoted repackaged applications are often free versions of apps that normally cost money.

7. Miscellaneous This class describes a number of miscellaneous research efforts that do not fit in any other class type.

6.2.3 Overview of (proposed) frameworks

In Table 6.1 we provide a systematization of noteworthy proposals, classified based on their purpose and characterized using the attributes discussed earlier.

Table 6.1: Overview of (proposed) frameworks

Category	Framework	Type: Attack	Type: Defense	Type: Analysis	Type: Detector	Technique: Dynamic	Technique: VMI	Technique: Syscalls	Technique: Method tracing	Technique: Taint tracking	Technique: Static	Technique: Decompiler	Technique: Weaving	Deployment: Android app	Deployment: Android OS	Deployment: Application	Availability	Summary
Static Analysis	ANDROGUARD [20]			×	×						×	×				×	open	ANDROGUARD is a popular comprehensive static analysis tool for Android applications. It can disassemble and decompile Dalvik Bytecode back to Java source code. Given two APK files, it can also compute a similarity value to detect repackaged apps or known malware. It also has modules that can parse and retrieve information from the app's `AndroidManfest.xml`. Due to its flexibility, it is used by some other (dynamic) analysis frameworks that need to perform some form of static analysis.
	APKINSPECTOR			×							×	×				×	open	APKINSPECTOR is a static analysis platform for Android application analysts and reverse engineers to visualize compiled Android packages and their corresponding DEX code.
	APKTOOL			×							×					×	open	APKTOOL is a tool for reverse engineering Android applications. It can decode resources to nearly original form and rebuild them into a new Android package after they have been modified. APKTOOL can be used to add additional features or extra support to existing applications without going through the original author. We use APKTOOL to extract and rebuild existing applications during the bytecode weaving process described in Section 4.4.
	DEDEXER			×							×	×				×	open	DEDEXER is a disassembler tool for DEX files. It reads DEX files and converts them into an 'assembly-like' format which is largely influenced by the JASMIN syntax[2].
	DEXTER			×							×	×				×	web	DEXTER is a web application designed for static analysis of Android applications. Its features are comparable with those of ANDROGUARD and APKinspector but it has some additional collaboration functionality to ease knowledge sharing among multiple researchers.
	DARE [49]			×							×	×				×	open	DARE is a project which aims to enable static analysis on Android applications by retargeting them to traditional `.class` files.
	DED [27]			×							×	×				×	closed	DED converts Android applications in DEX format to traditional `.class` files which can be processed by existing Java tools. It was superceded by DARE.
	DEX2JAR			×							×	×				×	open	DEX2JAR is a tool for converting Android's DEX formatted files to Java bytecode. Given an APK, DEX2JAR can convert it directly to a `.jar` file and vice versa. We use DEX2JAR in our bytecode weaving process outlined in Section 4.4.
	JEB			×							×	×				×	paid	JEB is a commercial flexible interactive Android decompiler. It claims to be able to directly decompile Dalvik bytecode to Java source code. It should also be able to disassemble an APK's contents so that users can view the decompressed manifest, resources, certificates, etc. Although its webpage contains comparisons of JEB against DEX2JAR, it does not mention the decompiler functionality of ANDROGUARD, which also directly converts Dalvik to Java source, instead of going through Java bytecode.
	SMALI			×							×	×				×	open	SMALI/BAKSMALI is an assembler/disassembler for the DEX file format. Like DEDEXER, it's syntax is loosely based on the JASMIN syntax.
	RADARE2			×							×	×		×		×	open	RADARE2 is an open source reverse engineering framework which provides a set of tools to disassemble, debug, analyze, and manipulate binary Android files. An official app is available for download from Google Play and can be used to dissect installed applications.
	JULIA [53]			×							×					×	paid	JULIA is a static analyzer for Java bytecode to perform formally correct analysis based on abstract interpretation and can be used to automatically find bugs and flaws in existing applications. Recent efforts were done to include support for Dalvik bytecode.

continued . . .

[2] JASMIN is an assembler for the Java VM: `http://jasmin.sourceforge.net`

Category	Framework	Type: Attack	Type: Defense	Type: Analysis	Type: Detector	Technique: Dynamic	Technique: VMI	Technique: Syscalls	Technique: Method tracing	Technique: Taint tracking	Technique: Static	Technique: Decompiler	Technique: Weaving	Deployment: Android app	Deployment: Android OS	Deployment: Application	Availability	Summary
Dynamic Analysis	AASandbox [7]			×	×	×		×			×				×	×	NA	AASandbox uses both static and dynamic analysis to analyze Android programs to automatically detect suspicious applications. Static analysis scans the package for malicious patterns without installing it, while the dynamic analysis implementation of AASandbox is placed in kernel space and hijacks systems calls for further analysis. It uses the monkey exerciser to generate random user input.
	Andrubis [42]			×	×	×	×	×	×	×	×			×	×	×	web	Andrubis is an extension of the Anubis service: a platform for analyzing unknown applications. To the best of our knowledge, Andrubis was the first dynamic analysis platform for Android applications that has been made publicly available as an easy-to-use web application in May 2012. Users can submit Android applications and receive a detailed report including a maliciousness rating when analysis finished.
	AppsPlayground [56]			×		×		×	×	×	×				×	×	open	AppsPlayground is a framework that automates analysis of Android applications and monitors taint propagation (using TaintDroid), specific API calls and system calls. Its main contribution is a heuristic-based intelligent black-box execution approach to explore the app's GUI.
	CopperDroid [58]			×		×	×	×			×					×	web	CopperDroid uses VMI-based dynamic system call-centric analysis to describe the behavior of Android applications. It features a stimulation technique to improve code coverage, aimed at triggering additional behaviors of interest. Like Andrubis, it is available to the public as a web application where users can submit samples for analysis.
	DroidBox 1			×		×			×	×	×				×	×	open	DroidBox is a dynamic analysis platform for Android applications. It adds additional tracing code to the core libraries and uses TaintDroid to detect private data leakages. A Python script is responsible for automating analysis and interpreting output results.
	DroidBox 2			×		×			×	×	×		×		×	×	open	A recent update of DroidBox introduces APIMonitor: a tool that can rewrite existing applications to add monitoring code for specific API calls. This removes the need of having to port DroidBox to newer Android versions, as the core library modifications are now no longer necessary. APIMonitor rewrites Dalvik bytecode directly, in contrast to our approach outlined in Section 4.4.
	DroidScope [72]			×		×	×	×	×	×						×	open	DroidScope is a fine-grained dynamic binary instrumentation tool for Android that rebuilds two levels of semantic information: OS and Java. It provides an instrumentation interface which can be used to write plug-ins. API tracing, native instruction tracing, Dalvik instruction tracing and taint tracking plug-ins have already been implemented. DroidScope works entirely on the emulator level and requires no changes to the Android sources.
	Mobile-Sandbox [64]			×		×			×	×	×				×	×	web	Mobile-Sandbox is a system designed to automatically analyze Android applications using both static and dynamic analysis. It uses DroidBox to keep track of Java code execution and a ported version of `ltrace` to track native code. TaintDroid is used to track private data leakages.
	SandDroid*			×	×	×			×	×	×				×	×	web	SandDroid is an automatic Android application analysis sandbox. It uses Androguard and DroidBox to perform dynamic analysis and its reports come with a risk score to indicate potential threats.
	ForeSafe*			×	×	×					×			×	×	×	web	ForeSafe Mobile Security combines static analysis with a fully automated server-side dynamic analysis system. ForeSafe parses reconstructed source code for patterns of known bad or suspicious behavior to detect malware. Dynamic analysis is used to reveal the use of obfuscated strings and Internet activity. An official Android application is available which allows users to scan installed applications on their device or to upload them to ForeSafe's cloud solution for remote analysis.

continued ...

Category	Framework	Type: Attack	Type: Defense	Type: Analysis	Type: Detector	Technique: Dynamic	Technique: VMI	Technique: Syscalls	Technique: Method tracing	Technique: Taint tracking	Technique: Static	Technique: Decompiler	Technique: Weaving	Deployment: Android app	Deployment: Android OS	Deployment: Application	Availability	Summary
Dynamic Analysis	JoeSecurity*			×	×	×			×		×	×			×	×	web	JoeSecurity analyzes APKs in a controlled environment and monitors the runtime behavior of the APK for suspicious activities. It uses an instrumentation engine to repack Android applications with instrumentation code. Dynamic analysis results are later mapped against statically obtained disassembly code. It also stimulates broadcast receivers by simulating specific intents. It uses *hybrid code analysis* to detect and classify malicious behavior within an APK.
Dynamic Analysis	Google Bouncer* [44]			×	×	×	?	?	?	?	×	?	?		×	×	NA	In February 2012, Google announced Bouncer: a system for automated scanning of the Google Play Store for potentially malicious software. The service performs static analysis to scan apps for known malware, spyware and trojans while every application will also be installed on Google's cloud infrastructure to be simulated as how it will run on a real Android device.
Dynamic Defenses	Andromaly [63]		×		×	×								×			open	Andromaly is a host-based malware detection system that continuously monitors smartphone features and events. It applies machine learning to classify the app's behavior (time between keystrokes, CPU consumption, network traffic, etc.) as normal (benign) or abnormal (malicious).
Dynamic Defenses	Apex [47]		×			×									×		open	Apex is an extension of the Android platform which makes it possible to enable or disable permissions of newly installed applications on an individual basis.
Dynamic Defenses	AppFence [35]		×			×									×		open	AppFence is an extension of the Android platform which makes it possible to allow newly installed applications access to only specific components (e.g., use Internet access only to connect to known advertisement hosts, use fake contact information, etc.).
Dynamic Defenses	Aurasium [71]		×			×							×			×	web	Aurasium can harden android apps so that the user is prompted whenever the app attempts suspicious actions (read privacy info, send SMS, etc.).
Dynamic Defenses	ConUCON [4]		×			×									×		NA	ConUCON extends the existing Android security mechanisms to implement a policy enforcement framework which enables users to grant permissions in a fine-grained manner and to support revocations and modifications on an app's permissions at runtime.
Dynamic Defenses	CrowDroid [11]		×		×	×		×						×			NA	CrowDroid uses a form of crowdsourcing to avoid the spread of detected malware to a larger community. An Android application is responsible for monitoring the device's system calls and forwards this data to a remote server. There, analysis is done to detect malware based on the system calls made.
Dynamic Defenses	I-ARM-Droid [19]		×			×							×			×	NA	I-ARM-Droid rewrites android apps so that they contain a reference monitor to allow users to apply security policies for a set of security sensitive API methods.
Dynamic Defenses	Kirin [24]		×			×									×		open	Kirin is a logic-based tool for Android that ensures permissions needed by apps are met by global safety invariants.
Dynamic Defenses	MockDroid [6]		×			×									×		open	MockDroid is a modified Android platform that allows users to control the permissions of an application at runtime.
Dynamic Defenses	Paranoid Android [54]		×		×	×		×		×					×		NA	Paranoid Android runs replicas of a user's device in the cloud on which security checks are performed. These include dynamic analysis to detect certain types of zero-days attacks (using taint tracking analysis), system call anomaly detection and anti-virus file scanning.
Dynamic Defenses	Pegasus [16]		×			×							×			×	NA	Pegasus is a system that uses permission event graphs to automatically detect sensitive operations being performed without the user's consent.
Dynamic Defenses	Quire [21]		×			×									×		NA	Quire is a modified Android platform that tracks the call chain of on-device IPC allowing an app the choice of operating with the reduced privileges of its callers or exercising its full privilege set by acting explicitly on its own behalf.
Dynamic Defenses	Saint [50]		×			×									×		NA	Saint is a modified Android platform to provide apps the utility to control to which other apps their interfaces/capabilities are granted.

continued . . .

Category	Framework	Type				Technique								Deployment				Summary
		Attack	Defense	Analysis	Detector	Dynamic	VMI	Syscalls	Method tracing	Taint tracking	Static	Decompiler	Weaving	Android app	Android OS	Application	Availability	
Dynamic Defenses	TaintDroid [26]		×		×	×				×					×		open	TaintDroid is a system-wide dynamic taint tracking and analysis system for simultaneously tracking multiple sources of sensitive data. It provides realtime analysis by leveraging Android's virtualized execution environment while introducing a 14% performance overhead.
	TISSA [79]		×			×									×		NA	TISSA (Tamming Information-Stealing Smartphone Applications) empowers users to flexibly control in a fine-grained manner what kinds of personal information will be accessible to an application. It's policies can be dynamically adjusted at runtime.
	XManDroid [10]		×		×	×									×		NA	XManDroid extends Android's monitoring mechanism to detect and prevent application-level privilege escalation attacks at runtime based on a system-centric system policy.
	YAASE [59]		×			×				×					×		NA	YAASE is an Android security extension that supports fine-grained access control policies. YAASE uses TaintDroid to enforce security decisions on how data has to be disseminated within the device (app to app) or the outside world (through Internet).
Static Defenses	Cerbo et al. [13]				×						×					×	NA	Cerbo et al. present a methodology for mobile forensics analysis to detect malicious applications. The methodology relies on the set of permissions exposed by each application.
	DroidChecker [15]				×					×	×					×	NA	DroidChecker is an automated analysis system to detect capability leaks in new Android applications and to find out how prevalent they are in existing apps. It uses interprocedural control flow graph searching and static taint checking to detect exploitable data paths.
	DroidRanger [80]				×						×					×	NA	DroidRanger is a system for detecting both new and already known Android malware. It uses permission-based behavioral footprinting to detect new samples of known Android malware families and heuristics-based filtering to identify certain inherent behaviors of unknown malicious families.
	Mann and Starostin [45]		×		×						×					×	NA	Mann and Starostin propose a framework to check whether the Dalvik bytecode of a given application conforms to a specific privacy policy. It detects privacy leaks like TaintDroid while using static information flow analysis.
	MAST [14]				×						×					×	NA	MAST is a Mobile Application Security Triage architecture that uses statistical analysis to direct scarce malware analysis resources towards the apps with the greatest potential to exhibit malicious behavior.
	RiskRanker [33]				×						×					×	NA	RiskRanker is a scalable automated system to analyze whether an app exhibits dangerous behavior (e.g., launching root exploits or sending background SMS messages).
	Sarma et al. [60]		×								×					×	NA	Sarma et al. compare an app's requested permissions against the permission set of similar applications of the same category. This way, they can inform users whether the risks of installing an application is commensurate with its expected benefit.
	SCanDroid [31]		×		×						×					×	open	SCanDroid extracts security specifications from an application's manifest and then applies data flow analysis using the app's source code to reason about the consistency of the specifications. Its intended use is comparable with TaintDroid.
	Stowaway [28]		×		×						×					×	NA	Stowaway detects over-privileged Android applications by comparing the set of used API calls against the API calls requested via the `AndroidManifest.xml`.
	Whyper [52]		×								×					×	NA	Whyper focuses on permissions for a given app and examines whether the app descriptions provides any indication for why the app needs the permissions.
Attacks	Andbot [69]	×												×			NA	Andbot is a design of a mobile botnet exploiting a novel command and control (C&C) strategy named URL flux. The bot would have desirable features including being stealthy, resilient and low-cost which promises to be appealing for bot masters.
	Davi et al. [18]	×												×			NA	Davi et al. demonstrate a privilege escalation attack on Android using a ROP attack on a higher privileged application.

continued ...

Overview of (proposed) frameworks, continued

Category	Framework	Type: Attack	Type: Defense	Type: Analysis	Type: Detector	Technique: Dynamic	Technique: VMI	Technique: Syscalls	Technique: Method tracing	Technique: Taint tracking	Technique: Static	Technique: Decompiler	Technique: Weaving	Deployment: Android app	Deployment: Android OS	Deployment: Application	Availability	Summary
Attacks	DroidChameleon [57]	×														×	NA	DroidChameleon is a systematic framework with various transformation techniques to measure the resistant of anti-virus vendors against various common obfuscation techniques. The paper concludes that none of the ten popular commercial anti-malware applications for Android are resistant against common malware transformation techniques.
	Orthacker et al. [51]	×												×			NA	Orthacker et al. illustrate how two apps with different permission sets can complement each other by using arbitrary communication channels to use each others capabilities (privilege escalation or confused deputy attack).
Repackaging	AppInk [76]		×		×						×					×	NA	AppInk uses a dynamic graph based watermarking mechanism to detect and deter further propagation of repackaged apps. It takes the source code of an app as input to automatically generate a new app with a transparently-embedded watermark and a manifest app. The manifest app can later be used to reliably recognize the embedded watermark.
	DroidMOSS [75]				×						×					×	NA	DroidMoss is a similarity measurement system that applies a fuzzy hashing technique to effectively localize and detect repackaged apps.
	PiggyApp [77]				×						×					×	NA	PiggyApp is a fast and scalable approach to detect 'piggybacked' apps: legitimate apps that are repackaged by malicious authors with destructive payloads.
Miscellaneous	AndroidRipper [2]			×		×										×	open	AndroidRipper is an automated technique that tests Android apps via their GUI and is based on a user-interface driven ripper that automatically explores the app's GUI with the aim of exercising the application in a structured manner.
	Hu and Neamtiu [36]			×		×									×	×	NA	Hu and Neamtiu present techniques for detecting GUI bugs by automatic generation of test cases, feeding the application random events, instrumenting the VM, producing log/trace files and analyzing them post-run.
	PScout [3]			×							×					×	open	PScout is a tool that extracts the permission specifications from the Android OS source code using static analysis. This information is used to overcome the incomplete documentation of Android's permission system.
	RobotDroid [73]		×												×		NA	RobotDroid is a malware detection framework that uses SVM active learning algorithm.
	SmartDroid [74]			×		×					×					×	NA	SmartDroid is a prototype system that shows how to automatically and efficiently detect an app's UI-based trigger conditions that are required to expose the app's sensitive behavior.

* These analysis platforms are closed source and do not come from scientific efforts. This means that we could not obtain detailed information about their internal functioning.

6.3 Dynamic Analysis Platforms

In this section, we further explore existing dynamic analysis platforms and compare their implementations against TraceDroid.

6.3.1 AASandbox

In October 2010, Bläsing et al. were the first to present a dynamic analysis platform for Android applications: AASandbox (Android Application Sandbox) [7]. It uses static analysis to scan software for malicious patterns and performs dynamic analysis to intervene and log low-level interactions with the system for further analysis by means of a loadable kernel module developed to obtain system call logs. AASandbox uses a system call footprinting approach for detecting suspicious applications. Unfortunately, there were no known Android malware samples available at the time to evaluate this technique. AASandbox seems to be unmaintained nowadays.

Compared to AASandbox, TraceDroid is implemented on a higher abstraction layer, namely the Dalvik VM instead of the Linux kernel. This allows TraceDroid to retrieve great detail on executed Java components, while missing any native code execution paths. By using the `strace` utility, however, we obtain a similar overview of executed system calls. We already use analysis output to detect suspicious activity.

6.3.2 TaintDroid

Also in October 2010, Enck et al. presented TaintDroid: a modified Android OS keeping track of taint propagation at runtime to detect privacy leaks [26]. Over time, it has been adopted as a valuable addition by many subsequent research proposals which aim to perform dynamic analysis on Android applications. TaintDroid is implemented as a modification of the Dalvik VM and thus cannot track taint within native code.

TaintDroid does not come with a set of scripts or applications to allow automated analysis and stimulation of unknown applications and is thus quite different compared to our TraceDroid platform. It does also not keep track of any specific method invocations. What we could do, however, is extending TraceDroid in such a way that it also keeps track of field operations, and use this data to implement taint tracking functionality as a post-processing plug-in.

6.3.3 DroidBox

DroidBox was developed by Patrik Lantz as part of Google Summer of Code (GSoC) 2011[3]. It combines TaintDroid with modifications of Android's core libraries. The modified Android OS of DroidBox logs the following events during runtime of an application:

- File read and write operations.
- Cryptography API activity.
- Opened network connections.
- Outgoing network traffic.

[3] `http://www.honeynet.org/node/744`

- Information leaks through network, files or SMS messages (using TaintDroid).
- Attempts to send SMS messages.
- Phone calls that have been made.

It also provides visualization of analysis results and automated app installation and execution.

TraceDroid differs from DroidBox in that our implementation traces *all* method invocations, including those occurring within an application, while DroidBox looks only at a small subset of API calls of which the developers think are interesting. Our approach is beneficial, for example if malware authors use third-party cryptographic libraries instead of the hooked APIs to encrypt data. DroidBox operates on the core library level, while TraceDroid is integrated at the Dalvik bytecode interpreter. This makes TraceDroid a more suitable platform for detailed analysis of unknown applications. On top of that, our TraceDroid Analysis Platform performs a more fine-grained level of simulation techniques when compared with DroidBox.

A second version of DroidBox was developed by Kun Yang as part of GSoC 2012 and introduces an APIMonitor that uses bytecode rewriting instead of core library modifications[4]. While this is a good approach to overcome the need of continuously upgrading DroidBox to newer Android versions, rewriting applications breaks an app's original signature which can easily be detected at runtime. Also, as with the first DroidBox release, the APIMonitor can only monitor core API calls which results in a less comprehensive set of traced method invocations than TraceDroid achieves.

Since DroidBox was the first openly available dynamic analysis platform for Android, it has been used as a base system by many other dynamic analysis platforms including Andrubis, Mobile-Sandbox, and SandDroid.

6.3.4 Bouncer

In February 2012, Google announced Bouncer [44]. It is stated that every application that is offered for download in the Google Play Store is run on Google's cloud infrastructure and gets simulated as if it was running on an Android device. Since Bouncer is used to protect the Google Play Store from malicious applications, only sparse information was provided about its internal functioning.

During Summercon in June 2012, however, Oberheide and Miller presented their efforts on dissecting Bouncer [**?**]. Using a C&C application that connects back to their local machine, they determined that Bouncer runs dynamic analysis on applications for 5 minutes. They could setup a connect-back shell to communicate with the application under investigation[5] and were able to obtain more detailed information on the environment used by Bouncer to run analysis.

In October 2012, Google introduced an application verification service. With the release of Android 4.2, the `ACTION_PACKAGE_NEEDS_VERIFICATION` broadcast was introduced, used by the OS to verify newly installed applications and check them for known malware. During installation, the OS sends information about

[4] http://www.honeynet.org/node/940

[5] http://www.youtube.com/watch?v=ZEIED2ZLEbQ

the app (including its package name and its SHA1 hash) and the device (its ID and IP address) to the Google cloud and requests a verification response. Although not confirmed, the Google cloud likely uses Bouncer results to make statements about the app's safety. A study performed by Xuxian Jiang shows that only 193 of the 1260 malgenome samples were detected as malicious by this verification scheme, indicating a detection rate of only 15.32%[6]

6.3.5 Andrubis

In June 2012, the International Secure Systems Lab released Andrubis: a dynamic analysis platform for Android applications [42]. Andrubis was the first to offer a publicly available web interface where users can submit Android applications for dynamic analysis. When analysis is finished, it generates a XML report containing a behavioral and static analysis footprint of the requested application. Its first release was based on DroidBox's core library modifications and was built on top of Android 2.1. Andrubis was later updated to run under Android 2.3.4 and uses VMI to intercept system calls made by native code execution.

With the integration of TraceDroid, Andrubis has become a very extensive dynamic analysis platform that does not only track taint propagation to detect privacy leaks, but also records invoked system calls and comprehensive Java method traces. Due to our collaboration efforts, Andrubis has adopted most of TraceDroid's functionality.

6.3.6 DroidScope

First presented in August 2012, DroidScope is a comprehensive dynamic binary instrumentation tool for Android based on VM introspection [72]. It reconstructs Dalvik instruction traces and this could, in theory, be used to obtain the same results as we currently obtain with TraceDroid. The key difference between TraceDroid and DroidScope, however, is the fact that DroidScope is bound to an emulator, while TraceDroid may run on actual hardware. Malicious applications can detect the use of an emulator and may decide not to start malicious activities in this scenario [55].

6.3.7 AppsPlayground

In February 2013, Rastogi et al. introduce AppsPlayground. Like previously discussed frameworks, AppsPlayground monitors taint propagation using TaintDroid, and traces specific Java API and system calls. Its main contribution is an improved monkey exerciser-like execution approach to explore application's GUIs. The latter is something we would like to add to our TraceDroid Analysis Platform as well, as a mean to increase code coverage.

6.3.8 Mobile-Sandbox

Mobile-Sandbox was released in March 2013 and is very similar to Andrubis in that it is based on DroidBox and uses TaintDroid to track taint propagation [64]. Instead of VMI to trace system calls, however, Mobile-Sandbox

[6] `http://www.cs.ncsu.edu/faculty/jiang/appverify`

uses a ported version of `ltrace` to trace native library invocations. This `ltrace` binary may be a valuable addition to our TraceDroid analysis platform. Like Andrubis, Mobile-Sandbox also allows users to submit Android applications for analysis via a web application.

6.3.9 CopperDroid

Finally, in April 2013, Reina et al. introduced CopperDroid [58]. The mechanisms used in this system are similar to DroidScope as it also uses VMI to collect system call information about analyzed applications. Reina et al. argue, however, that CopperDroid points out how their system call-centric analysis and stimulation techniques can comprehensively expose Android malware behaviors. CopperDroid also comes with a web interface where users can submit unknown applications for analysis.

6.3.10 Closed frameworks

A number of additional dynamic analysis platforms have been implemented and made available to the public via web applications. These frameworks, however, come with very little documentation on how they operate which makes it hard to make statements on any new approaches used by these implementations. It is likely that these platforms use (modified versions of) existing tools like DroidBox, TaintDroid and Androguard to complement their dynamic analysis engine. This is confirmed on SandDroid's webpage, which states that it is powered by both DroidBox and Androguard[7]. Example output reports of both ForeSafe[8] and JoeSecurity[9] suggest that these platforms use a combination of existing tools as well.

[7] `http://sanddroid.xjtu.edu.cn`
[8] `http://www.foresafe.com`
[9] `http://www.apk-analyzer.net`

Chapter 7

Conclusions

In this chapter, we outline our conclusions regarding dynamic analysis of Android malware. We first propose a set of future research efforts in Section 7.1, followed by a conclusive overview of our contributions in Section 7.2.

7.1 Future Work

In this section, we first discuss some future research directions that relate to TraceDroid and the TraceDroid Analysis Platform in particular in Sections 7.1.1 and 7.1.2. We outline notes regarding future research work on the field of Android malware in general in Section 7.1.3.

7.1.1 TraceDroid

Although our Android OS modifications that enables us to generate comprehensive method traces of Android applications are performing exceptionally well already, there are a number of additional features conceivable that will enhance the quality of analysis results.

Array unpacking

As with most programming languages, Java has the notion of array data structures which can be passed to or returned by invoked methods. TraceDroid does currently not 'unfold' these arrays but rather prints the array's `toString()` return value which often only contains the address in memory of where the data structure is located. While this may be sufficient to understand the control flow of an application, there are cases in which array unpacking reveals interesting details about an app's implementation.

One of the scenarios in which array unpacking is desired is when an app uses reflection. Reflection gives a developer the ability to examine and modify the structure and behavior of objects at runtime and can be used to obfuscate program code. Consider the example program depicted in Listing 7.1a of an Android application that uses reflection to lookup and invoke Android's `Log.i` function. The program's (partial) trace output is depicted in Listing 7.1b.

From Listing 7.1b we conclude that arguments passed to `android.util.Log.i()` are lost as they were stored in an `Object` array. If we could unpack the `String`

Listing 7.1: Android application using reflection

(a) Source

```
package com.example.example2;

import android.os.Bundle;
import android.app.Activity;
import java.lang.reflect.Method;

public class MainActivity extends Activity {

 protected void onCreate(Bundle b) {
  super.onCreate(b);

  try {
   /* Search for Android log class */
   Class c = Class.forName("android.util.Log");

   /* Create a parameter type array for method i */
   Class[] targs = {String.class, String.class};

   /* Get method android.util.Log.i(String tag, String msg) */
   Method m = c.getMethod("i", targs);

   /* Create a parameter array */
   String[] args = {"Hello", "World"};

   /* invoke android.util.Log.i("Hello", "World") */
   m.invoke(null, (Object[]) args);

  } catch (Exception e) {
   e.printStackTrace();
  }
 }
}
```

(b) Trace output

```
...
public static java.lang.Class java.lang.Class.forName((java.lang.String) "android.util.Log")
return (java.lang.Class) "class android.util.Log"
public java.lang.reflect.Method java.lang.Class("class android.util.Log").
         getMethod((java.lang.String) "i", (java.lang.Class[]) "[Ljava.lang.Class;@40519d38")
return (java.lang.reflect.Method)
         "public static int android.util.Log.i(java.lang.String,java.lang.String)"
public java.lang.Object java.lang.reflect.Method(
         "public static int android.util.Log.i(java.lang.String,java.lang.String)").
         invoke((java.lang.Object) "null", (java.lang.Object[]) "[Ljava.lang.String;@4051bc10")
return (java.lang.Object) "13"
..
```

array at memory address `0x4051bc10`, we would find the original `["Hello","World"]` array.

Reflection is an effective methodology for malicious authors to protect their malware against static analysis: by encrypting strings that contain the Android API classes and methods and decrypting them at runtime, static analysis tools will fail to detect suspicious API invocations. The OBad sample is an example

of malware that uses reflection to hinder analysis [66].

It must be noted that above reflection issues could also be fixed by tracing Dalvik's core library: the `java.lang.reflect.Method.invoke()` implementation will eventually invoke the target method with the original parameter structure. We miss this invocation in our trace file since the invoked target method does not originate from the targeted application but from the core library, and, as outlined in Section 4.1.1, we do not trace function calls within the OS if they do not originate from the target app. Object unpacking would still be a valuable addition however, as attackers could otherwise go into stealth mode by packing their internal function parameters into array data structures.

Another interest of array unpacking lies in reconstructing byte sequences that are used during I/O operations. Whenever such operation is initiated, concerned data is often passed as a byte array. Unpacking these arrays will allow us to fully reconstruct I/O operations, like, for instance, the exact sequence of bytes that are written to disk during a `write` invocation.

To add array unpacking to TRACEDROID, we can use Dalvik's existing `ArrayObject` structure which represents a Java array in C. We would first have to cast variables containing arrays to an `ArrayObject` and then loop over it to print its contents.

Tracing field operations A more complex feature would be adding the capability to trace all interpreted Dalvik bytecode. Such feature would allow us to not only trace method invocations, but also reconstruct loops (`while`, `do...while`, `for`), keep track of operations involving operators (`=`, `+`, `-`, ...) and trace decision making statements (`if...then`, `if...then...else`, `switch`), all at runtime. With this, we could try to reconstruct original source code that gives us an ever better overview of how the app functions internally.

To add this feature to TRACEDROID, we would have to modify the Dalvik interpreter in such a way that each executed instruction is printed to one of our output files.

Porting The current version of TRACEDROID is built on top of Android 2.3.4 (codename Gingerbread), released in April 2011. Until recently the majority of Android devices was still running Gingerbread. The last few months, however, we see that Jelly Bean (Android 4.1.x – 4.3.x) is taking over this position with a combined market share of over 40%[1] (on August 1, 2013). Porting TRACEDROID to newer Android releases is important to maintain support for apps that make use of recently introduced API functions. A quick look at the Android 4.3 source code shows that no fundamental changes were made to Android's profiler implementation and that porting TRACEDROID will be relatively easy.

Prevent evasion techniques We mentioned earlier that applications can detect if they are executed within an emulated environment [48]. Malware authors can use detection techniques to decide not to start malicious activities whenever their application is analyzed within such surroundings. Some of the techniques can easily be obstructed by modifying specific system variables within qemu, however, preventing all of them will be near to impossible to accomplish[2].

[1] `http://developer.android.com/about/dashboards/index.html`
[2] `http://dexlabs.org/blog/btdetect`

We therefore propose a scheme wherein we use an actual device installed with a TraceDroid-based firmware image so that we can use real hardware to test questionable applications. As with porting, building the Android source for devices is a straightforward process, especially when using a Google's Nexus phone as target as instructions for building firmwares for these devices are listed on Android's webpage[3].

Aside from emulator evasion techniques, we also have to take time scheduled actions into consideration. To prevent automated detection during an analysis session, an app may decide not to start malicious activity until a certain amount of time has passed. We should look for possibilities to detect these scheduled events and try to somehow invoke the scheduled action anyway. We could, for instance, hook specific API calls and change alarm timings at runtime. This would, however, not prevent against a scheme wherein computational effort (e.g., an almost endless loop) is used to postpone an app's activity. More research is necessary to prevent against these type of evasion as well.

7.1.2 TraceDroid Analysis Platform

Our analysis platform that stimulates applications and is responsible for post-processing TraceDroid output could also be further improved. We now outline a number of further research directions involving our TraceDroid Analysis Platform.

Malware detection Using our extensive feature set plug-in, we would like to have another post-processing script that uses a machine learning technique to classify unknown applications and detect new malware. A preliminary test case consisting of about 300 malware and 200 benign applications already shows that we can obtain a detection rate of about 93% to 96% (measured as F-Score: combined true positives and true negatives), which is a good prospect for future work.

Code coverage An easy addition to the TraceDroid Analysis Platform would be the separation of code coverage results on a per package basis. Since third-party libraries are often packaged with a different package name than the main application, such separation would give a more precise insight in how our stimulations perform in terms of code coverage. Separation of code coverage results per package also gives analysts a better insight into which packages are of interest for further research: they could quickly dismiss known third-party libraries and focus solely on the app's implementation.

In addition, we also want to search for improved stimulation techniques to increase code coverage of our analysis platform in general. We would like to use the heuristic-based execution approach as outlined by Rastogi et al. in order to better stimulate an app's interfaces [56]. Another noteworthy approach would be using symbolic execution on the Dalvik bytecode. For this, projects like JavaPathFinder[4] are of particular interest.

[3] `http://source.android.com/source/building-devices.html`
[4] `http://babelfish.arc.nasa.gov/trac/jpf`

Taint tracking If we manage to implement full bytecode tracing in TraceDroid, we could use its output to reconstruct taint propagation during post-analysis. We could use this to built a system similar to TaintDroid that detects private data leaks, but also follows incoming data to detect malicious code execution. Storing all executed operations gives us the advantage to check taint propagation using two directions: forward and backward. Considering the example of detecting data leaks, forward taint checking starts tracking variables as soon as private data is first accessed. With backward taint tracking, we start at potential sink access operations (file write, network write, etc.) and follow the execution trace back to where data was first accessed. Using a combination of both normal and backward taint analysis, we may be able to better detect data leaks through implicit data flows.

Replaying Another feature that could be realized once full bytecode tracing for TraceDroid is implemented, is a replay module that uses the traced (or recorded) bytecode instructions to exactly replay an app's execution for a second time. Replay functionality may be of value to debug specific race conditions that are otherwise hard to reproduce. To the best of our knowledge, replaying Android applications has not been subject of research before. Related projects that can replay Java applications are Chronon[5] and RD/Recorder[6].

7.1.3 Other research directions

We also identify a number of essential Android malware related research directions that are not explicit extensions of our current work. We briefly discuss them in the following paragraphs.

SMS stealing protection On a defensive side, the Android OS needs a mechanism to protect users against SMS stealing malware, especially if mobile TAN codes for secure banking transactions are involved. While Google could implement such feature easily by modifying the OS internally, we believe that it is also possible to protect a user using regular applications. We already started working on a prototype application that runs entirely in user-space and detects applications that start network traffic shortly after a new SMS message containing mTAN-like content arrived.

Awareness A more psychology based study is desired to determine why users are tricked in installing malicious applications and how we can raise awareness among consumers. To the best of our knowledge, no mobile malware is currently capable of installing itself without a user's consent, which means that malware gets distributed using only social engineering techniques. It is important to understand the psychology of affected consumers so that we can direct research towards this particular problem.

[5] `http://chrononsystems.com`
[6] `http://www.replaysolutions.com/products/recorder`

7.2 Conclusions

The high pace of which mobile malware is being spread among Android devices, calls for a new approach to quickly analyze and detect previously unknown malware families. In Chapter 4, we have presented the TraceDroid Analysis Platform, a platform for automated dynamic analysis of unknown Android applications using a comprehensive method tracing scheme dubbed TraceDroid. For each application, the platform can generate a feature set containing suspicious activities triggered during the app's execution run. One can use these analysis results to detect new malware samples of known families or suspicious applications that require a more in-depth analysis and may belong to a previously unknown malware class. In any case, our proposed framework directs scarce analysis resources towards applications that have the greatest potential of being malicious.

In addition to our work relating to dynamic analysis of Android applications, we have also provided a comprehensive systematization of knowledge study in which we classify and summarize a large number of research projects focusing on Android security in general in Chapter 6.

7.2.1 TraceDroid

With TraceDroid, we have integrated a comprehensive method tracer into the existing Android OS source code. Upon each method invocation, we obtain the method's complete signature, its object's `toString()` representation (if any) and a list of parameters passed to the method. Upon return statements, we display the function's return value or, in case of an exception, the exception object that is thrown to the caller. For parameters or return values that are non-primitive, we invoke the object's `toString()` function in order to obtain a textual representation of the item in question. In addition, we keep track of a function's call depth and also include timestamps in our log output. Benchmark results show that our implementation gains an almost 50% speedup compared to Android's original profiler.

Although TraceDroid was originally developed to aid only malware or anti-virus researchers in reconstructing malicious instruction traces using dynamic analysis, its user base can be extended to include developers and reverse engineers as well. Developers can use TraceDroid as a replacement for the original Android profiler. TraceDroid is not only faster but also reveals a more detailed overview of invoked functions while omitting uninteresting internal OS invocations. Reverse engineers can use TraceDroid in combination with static analysis tools to quickly reverse an app's implementation.

7.2.2 TraceDroid Analysis Platform

The TraceDroid Analysis Platform is a framework for automated analysis of applications using dynamic analysis. We implemented a number of post-processing plug-ins including a code coverage computation script and a feature extraction tool. Using the code coverage plug-in, we conclude that our stimulation engine gains an average code coverage during dynamic analysis of about 33%. A preliminary study on detecting malware using the extracted features

show a detection rate of 93% – 96%. Our platform can thus be extended to quickly identify suspicious Android applications that are likely to be malicious.

In order to aid complementary manual analysis on TraceDroid's method trace output, we provide a sophisticated inspection tool that parses TraceDroid's output into Python objects and gives the user an interactive shell to perform in-depth analysis. Using this utility, an analyzer can generate a call graph of analyzed applications containing an overview of all invoked methods and their call chain, while they are grouped into clusters based on their belonging classes. The call graph helps analysts to quickly understand control flow paths within applications and aid further analysis. In Section 5.5, we demonstrated how our tool can be used to dissect a malicious application.

Our framework is highly flexible and allows analysts to write additional post-processing plug-ins using our provided interface. This gives analysts the power to add specific features based on TraceDroid's method trace output.

Bibliography

[1] *Kindsight Security Labs Malware Report - Q2 2013.* Alcatel-Lucent, Jul. 2013.

[2] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI Ripping for Automated Testing of Android Applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Sep. 2012.

[3] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Oct 2012.

[4] Guangdong Bai, Liang Gu, Tao Feng, Yao Guo, and Xiangqun Chen. Context-Aware Usage Control for Android. In *Proceedings of the 6th International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, Sep. 2010.

[5] Michael Becher, Felix C. Freiling, Johannes Hoffmand, Thorsten Holz, Sebastian Uellenbeck, and Christopher Wolf. Mobile Security Catching Up? Revealing the Nuts and Bolts of the Security of Mobile Devices. In *Proceedings of the 32nd Annual IEEE Symposium on Security and Privacy (S&P)*, May. 2011.

[6] Alastair R. Beresford, Andrew Rice, Nicholas Skehin, and Ripduman Sohan. MockDroid: trading privacy for application functionality on smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications (HotMobile)*, Mar. 2011.

[7] Thomas Bläsing, Leonid Batyuk, Aubrey-Derrick Schmidt, Seyit Ahmet Camtepe, and Sahin Albayrak. An Android Application Sandbox System for Suspicious Software Detection. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALWARE)*, Oct. 2010.

[8] Dan Bornstein. *Dalvik VM Internals.* Google I/O, May 2008.

[9] Stefan Brähler. Analysis of the Android Architecture, Oct. 2010.

[10] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, and Ahmad-Reza Sadeghi. XManDroid: A New Android Evolution to Mitigate Privilege Escalation Attacks. Technical report, Technische Universität Darmstadt, Apr. 2011.

[11] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-Based Malware Detection System for Android. In *Proceedings of the 1st Annual ACM CCS workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, Oct. 2011.

[12] *Over 1 billion Android-based smart phones to ship in 2017*. Canalys, Jun. 2013.

[13] Francesco Di Cerbo, Andrea Girardello, Florian Michahelles, and Svetlana Voronkova. Detection of malicious applications on Android OS. In *Proceedings of the 4th International Conference on Computational forensics (IWCF)*, Nov. 2011.

[14] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. MAST: Triage for Market-scale Mobile Malware Analysis. In *Proceedings of the 6th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, Apr. 2013.

[15] Patrick P.F. Chan, Lucas C.K. Hui, and S.M. Yiu. DroidChecker: Analyzing Android Applications for Capability Leak. In *Proceedings of the 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WISEC)*, Apr. 2012.

[16] Kevin Zhijie Chen, Noah Johnson, Vijay D'Silva, Shuaifu Dai, Kyle MacNamara, Tom Magrino, Edward Wu, Martin Rinard, and Dawn Song. Contextual Policy Enforcement in Android Applications with Permission Event Graphs. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2013.

[17] Eric Chien. Motivations of Recent Android Malware. *Symantec Security Response*, Oct. 2011.

[18] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *Proceedings of the 13th Information Security Conference (ISC)*, Oct. 2010.

[19] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications. *Mobile Security Technologies (MoST)*, May 2012.

[20] Anthony Desnosi and Geoffroy Gueguen. Android: From Reversing to Decompilation. In *Proceedings of Black Hat Abu Dhabi*, Dec. 2011.

[21] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium*, Aug. 2011.

[22] David Ehringer. The Dalvik Virtual Machine Architecture, Mar. 2010.

[23] William Enck. Defending Users against Smartphone Apps: Techniques and Future Directions. In *Proceedings of the 7th International Conference on Information Systems Security (ICISS)*, Dec. 2011.

[24] William Enck, Machigar Ongtang, and Patrick McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, Nov. 2009.

[25] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android Security. *IEEE Security & Privacy*, 7(1), Feb. 2009.

[26] William Enck, Peter Gilbert, Byung-Gon Chunn, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Oct. 2010.

[27] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, Aug. 2011.

[28] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android Permissions Demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2011.

[29] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steven Hanna, and David Wagner. A Survey of Mobile Malware in the Wild. In *Proceedings of the 1st Annual ACM CCS workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, Oct. 2011.

[30] *FortiGuard Midyear Threat Report*. Fortinet, Aug. 2013.

[31] Adam P. Fuchs, Avik Chaudhuri, and Jeffrey S. Foster. SCanDroid: Automated Security Certification of Android Applications. Technical report, University of Maryland, Nov. 2009.

[32] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2003.

[33] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MOBISYS)*, Jun. 2012.

[34] The Hindu. UAB computer forensics links internet postcards to virus. `http://www.hindu.com/thehindu/holnus/008200907271321.htm`, Jul. 2009.

[35] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. "These Aren't the Droids You're Looking For" Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, Oct. 2011.

[36] Cuixiong Hu and Iulian Neamtiu. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test (AST)*, May 2011.

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

[38] *Third Annual Mobile Threats Report*. Juniper Networks, Jun. 2013.

[39] Eran Kalige and Darrel Burkey. A Case Study of Eurograbber: How 36 Million Euros was Stolen via Malware, Dec. 2012.

[40] Ramnivas Laddad. *AspectJ in Action, Practical Aspect-Oriented Programming*. Manning Publications, 2003.

[41] Patrik Lantz. DroidBox, Feb. 2011.

[42] Martina Lindorfer. Andrubis: A Tool for Analyzing Unknown Android Applications. `http://blog.iseclab.org/2012/06/04/andrubis-a-tool-for-analyzing-unknown-android-applications-2`, June 2012.

[43] Ramon Llamas, Ryan Reith, and Michael Shirer. Apple Cedes Market Share in Smartphone Operating System Market as Android Surges and Windows Phone Gains, According to IDC. `http://www.idc.com/getdoc.jsp?containerId=prUS24257413`, Aug. 2013.

[44] Hiroshi Lockheimer. Android and Security. `http://googlemobile.blogspot.nl/2012/02/android-and-security.html`, Feb. 2012.

[45] Christopher Mann and Artem Starostin. A Framework for Static Detection of Privacy Leaks in Android Applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC)*, Mar. 2012.

[46] Denis Maslennikov. Zeus-in-the-Mobile — Facts and Theories. `http://www.securelist.com/en/analysis/204792194`, Oct. 2011.

[47] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Apr. 2010.

[48] Jon Oberheide and Charlie Miller. Dissecting the Android Bouncer, Jun. 2012.

[49] Damien Octeau, Somesh Jha, and Patrick McDaniel. Retargeting Android Applications to Java Bytecode. In *Proceedings of the 20th International Symposium on the Foundations of Software Engineering (SIGSOFT)*, Nov. 2012.

[50] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, Dec. 2009.

[51] Clemens Orthacker, , Peter Teufl, Stefan Kraxberger, Günther Lackner, Michael Gissing, Alexander Marsalek, Johannes Leibetseder, and Oliver Prevenhueber. Android security permissions - can we trust them? In *Proceedings of the 3rd International Conference on Security and Privacy in Mobile Information and Communication Systems (MOBISEC)*, Jun. 2011.

[52] Rahul Pandita, Xusheng Xiao, Wei Yang, Willieam Enc, and Tao Xie. WHYPER: Towards Automating Risk Assessment of Mobile Applications. In *Proceedings of the 22nd USENIX Security Symposium*, Aug. 2013.

[53] Étienne Payet and Fausto Spoto. Static analysis of Android programs. *Information and Software Technology*, 54(11), Jun. 2012.

[54] Georgios Portokalidis, Philip Homburg, Kostas Anagnostakis, and Herbert Bos. Paranoid android: versatile protection for smartphones. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*, Dec. 2010.

[55] Thomas Raffetseder, Christopher Krügel, and Engin Kirda. Detecting system emulators. In *Proceedings of the 10th Information Security Conference (ISC)*, Oct. 2007.

[56] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the 3rd ACM conference on Data and Application Security and Privacy (CODASPY)*, Feb. 2013.

[57] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. DroidChameleon: Evaluating Android Anti-malware against Transformation Attacks. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, May 2013.

[58] Alessandro Reina, Aristide Fattori, and Lorenzo Cavallaro. A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors. In *Proceedings of the 6th European Workshop on System Security (EUROSEC)*, Apr. 2013.

[59] Giovanni Russello, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich. YAASE: Yet Another Android Security Extension. In *Proceedings of the 3rd International Conference on Social Computing (SocialCom)*, Oct. 2011.

[60] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android Permissions: A Perspective Combining Risks and Benefits. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies (SACMAT)*, Jun. 2012.

[61] Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Jan Clausen, Kamer Ali Yüksel, Osman Kiraz, Ahmet Camtepe, and Sahin Albayrak. Enhancing Security of Linux-based Android Devices. In *Proceedings of the 15th International Linux System Technology Conference*, Oct. 2008.

[62] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google Android: A Comprehensive Security Assessment. *IEEE Security & Privacy*, 8(2), Apr. 2010.

[63] Asaf Shabtai, Uri Kanonov, Yuval Elovici, Chanan Glezer, and Yael Weiss. "andromaly": a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1), 2012.

[64] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-Sandbox: Having a Deeper Look into Android Applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC)*, Mar. 2013.

[65] *TrendLabs 2Q 2013 Security Roundup*. Trend Micro, Aug. 2013.

[66] Roman Unucheck. The most sophisticated Android Trojan. `http://www.securelist.com/en/blog/8106/The_most_sophisticated_Android_Trojan`, Jun. 2013.

[67] Timothy Vidas, Daniel Votipka, and Nicolas Christin. All Your Droid Are Belong to Us: A Survey of Current Android Attacks. In *Proceedings of the 5th USENIX Workshop on Offensive Technologies (WOOT)*, Aug. 2011.

[68] Christina Warren. Google Play Hits 1 Million Apps. `http://mashable.com/2013/07/24/google-play-1-million`, Jul. 2013.

[69] Cui Xiang, Fang Binxing, Yin Lihua, Liu Xiaoyi, and Zang Tianning. Andbot: Towards Advanced Mobile Botnets. In *Proceedings of the 4th USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET)*, Mar. 2011.

[70] Pan Xiaobo. dex2jar, Oct. 2012.

[71] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Security Symposium*, Aug. 2012.

[72] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Security Symposium*, Aug. 2012.

[73] Min Zhao, Tao Zhang, Fangbin Ge, and Zhijian Yuan. RobotDroid: A Lightweight Malware Detection Framework On Smartphones. *Journal of Networks*, 7(4), Apr. 2012.

[74] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. SmartDroid: an Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Proceedings of the 2nd Annual ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, Oct. 2012.

[75] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy (CODASPY)*, Feb. 2012.

[76] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. AppInk: Watermarking Android Apps for Repackaging Deterrence. In *Proceedings of the 8th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, May 2013.

[77] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, Scalable Detection of "Piggybacked" Mobile Applications. In *Proceedings of the 3rd ACM Conference on Data and Application Security and Privacy (CODASPY)*, Feb. 2013.

[78] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 33rd Annual IEEE Symposium on Security and Privacy (S&P)*, May 2012.

[79] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing (TRUST)*, Jun. 2011.

[80] 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 *Proceedings of the 19th Annual Network & Distributed System Security Symposium (NDSS)*, Feb. 2012.

Appendices

Appendix A

Sample Set

Table A.1 enumerates the 250 MD5 hashes of the benign sample set used in this thesis. For each sample, the corresponding package name is listed.

Table A.1: Benign sample set

MD5 hash	Package name
03aaf04fa886b76303114bc430c1e32c	com.lftechs.tictactoe.free
05736b33323671617d51a785d15771dd	com.alcove.cch
08047cfe0af0abfd91e0654fea03b329	com.yschi.MyAppSharer
0865a0b86f51cd8f17e35d68db4340e0	com.logmein.joinme
09d23c757d9d53b86c6d64e1a37ec892	com.iviewsoftware.pballFree
0abf92e77bb83c8b8419a2653c2d0027	com.oziapp.BirdHuntingLite
0f0b9ff3a870435b36d91471b95f47dc	com.putitonline.da
109e2b0ece8353cd63af25e922064128	com.ninja.ume.u1310978874968
1123a5ae27124deb5457b44d2bdc44de	com.etc.android.ecs
123858dcd9ca21c39a345ce0aa640301	com.app.grandag.trackchecker
128a971ff90638fd7fc7afca31dca16b	com.appspot.yongSubway_NZ
12b7a4873a2adbd7d4b89eb17d57e3aa	com.brightai.middlesboroguide
12dc6496fdd54a9df28d991073f26749	com.via3apps.sensacio142
1390e4fecca9888cdf0489c5fe717839	com.rpg90.seasons_cn
159958c76f9420d6775c5e54f7927e83	dalmax.games.arcade.jewels
15b241c73da676396fe73b26f5d89cf9	com.mycity247.mycity247
173fecb584d89da296fa0d0f5fb9a361	com.dogmac.F1News
1896912ab21677abb6f2031ccc687275	com.dreamtree.graffiti3dwallpapers
1b2ce824eec68a130f87f473a06aa16e	net.thekillerapp.superbonus
1b49f3b87d320fee118888dc909cda57	com.allucanapp.wine.free
1cd841e49559c033fab5712c33a8c089	com.demansol.lostinjungle.lite.activities
1ce477618558d152b4098c06b81e3dbf	com.picobrothers.vibrate
1f7096875a52c8ff4298b647fd440ac7	com.CastleApp.NutritionTips
1f7dcb54d2f961ba9573ddf43410ba76	mem.usage
1fb62027840784f071c19ae564d8a5bd	com.bovello.autoprofileswitcher
22491ea22fe391043a4b69990c363320	com.kauf.jokes.account1.FunnyAnimalJokes
235bbed0eb7ef2ca42aaa51dd218c7c2	com.klervi.velivert
262d6f79f3ca4946b8873fe81dda1ba4	de.nuromedia.android.talkingfriends_Free
26832755a721b66ecc76b3c0bf34203c	com.xxt.megoogle
27a53676e4f7326a74025f11b7c27e8b	bdmobile.android.app
287fa8f0a67688556307fb10a33720ab	forzaCesena.apexnet.it
298a6843d024992202696c9428f6fcdb	com.lcb.android.book.TheGirlwiththeDragonTattoo
29bd6d7d71d5816bae98f5fb4b962cb7	com.fltom.FLDerrickRose
2a025fab3a79892fadad96009cd473ee	com.zyquest.radio.kissfmlite
2a2feb80905eab1c6a35589b2d5c22e2	xdebugx.partyLightA

Benign sample set, continued

MD5 hash	Package name
2af10ac63bc5a785b1022f7fbeb4d319	com.agargroove.zeitungenatfree
2c064012644066a5a31c3b0cb7ba67f6	org.majoobi.App.medicalnw
2d091785088eac16b241a2d7f44a152a	com.boa.sheeptone
2e346c123e49b413ca7f3cf4823a1edf	com.melicoapp.br.GuiaUpFree
2e76cffc61f9bb33e00a7674d2eba9bf	com.wynonnajudd.droidradio
2f3115705af5b30148e597ab42388068	com.crwdpower.parkpatrol
3049c030821af3dac9946c70d057cdd9	laststand.lite
316cb193debbdb1ddf1f438cd24aeeab	teq.QCustomShortcut
316d6281449251375b3fbebc3553274b	psv.apps.expmanager
33ac8dc386342eb7ad42fb3b58b55042	com.kodexo.sheepbox
34c7d74f7210484375a9d1bb8e62f950	com.codeiv.PhotoBook.Free
357244a37e999de6c06bea26b1ae0e35	com.valleydevteam.Sprueche
35bf18b1eee1bf3ea47b0345f58c2eb5	com.theojoe.holidays
35db9c855f5a6457f616bdfa46b2a3b6	com.bzyg.diyicideqinmijiechu
35ff1318ab86b7ab5826c2f996f001fb	plat.wallpaperlucky041706
365639e6d6a3f688a67a18f045dbf9aa	com.gadgetmvp.gadget_D39FB1CF_5FFC_B2AB_7EC6_871233182BC0
36c9a20f844a94cbe22b1cb577ed454a	zureus.amazon.buzz.gifts
3712f6ae08467be6ca3e9e5ba7005c87	com.warting.blogg.wis_psvitalive_feed_nu
3731ae94aa3c6b8dd9fb13410c392805	com.nwave.android.CapViewer
37b8467ede1dc7e9ccba110536e51b42	fifteen.puzzle
37eacdc7366403eac3970124c3a3fc32	com.omgbutton
3a11d47f994ec85cfeff8e159de46c54	ynd.tapmadness
3a7764ad6a4fbd4b9b8a192c11c41034	jp.yyc.game.jumpman
3bba568226b7d8f997d97befc2981d7a	com.kidsfun.matching.starjewels
3be20a7db30d3e5de6f72449a64aeeb2	DroidApp.RoamingBillboard
3c142c03c1e8c19fe951a8eab32fb1f2	com.quoord.androidcommunity.activity
3d63a65f2e7cb762b40fd1ff43671d7e	moon.wallmar24m
3d6a85790de7cec5d1e35514d220e4f6	wall.bt.wp.P108
3eba106e83efef1fa9ba09b59debcc3f	vistaworks.RoyalGorgeTravel
3f67e2e2d7a97b7717ebd8a0a9f15f1c	com.shimada.biorhythm
3f796196f7bf6e1b4190fdf7cc4eb843	com.drinkowl
3fd054911ab1d99c44348b8208332caa	com.yc360.college.smsu
40082ae16ffc68f2dd64bc5fb5cf5c71	com.dreamstep.wCalling
400e7f5b4549c5d05a4243b3c4a36525	spring.wallpaper031909
409d369c3cedc337e57cd5d80a600459	com.zlango.zms
41b9a5230978b86acf945d8560e2b573	com.beautifulgirl.sexyleg1
426d7919384bc78369513dc865186b30	com.monkeyfly.android.bubble.DominoTwinkleP2
42cc83bb1875dff17e623edb9ef4a505	com.jamesarchuleta.PortalSoundBoard
44c2baedb297e6fd2f80fe60a8f47cea	mobi.news.minnesotatwins
450cfb30ce3215893b7ed8ec61e14811	org.muth.android.trainer_demo_pt
455b4d9d3112c4df996462195cdf01b5	com.sport.quiz.adzoone
4a996ef13068fa1c32de54dfa264fa97	com.crossfield.TheCrazyUFO
4c05e692d55616a0c5423b5ca25697ee	com.mnemotime.android
4e26abc0d66ac6230180a7ec4f17f5d9	com.electricsheep.boussole
4e4a1e5bc3366e559c2052d76352ac72	personal.jhjeong.app.batterylite
4ec805846b0de93483c80d5a354bbe5a	org.thibault.android.buzzphrase
4eefe607b18a89d82f5c778523ef4d8d	silly.walljun23j
5292488dc10ee3e0c46cd91275994125	com.gamesoul.combadge
56036dd4c63cd319bc523a9010288f19	com.androidbeans.techcrunch
56228df12dabd96434a3f05187a15d6f	com.tesyio.puzzle.numplace.vol1w
584621b557d913001d21eceb7a62e0ef	com.advisorlynx.mobileadvisor.portfolio
58766f54865ff0138dd10b4f3777b7c5	com.dailystats.mlb.bluejays
59f37665e96d14fe7c2f8221af089cd7	com.telemaque.travel.discover_canada
5ac46c265a27dc1be07ee4354da54524	com.hunghom.HDHHTHellokittyI
5bf2caabe9b0b5465eb553927f71e174	com.softmimo.android.connectfourfreeversion
5c040afa41c19ffdb9a418771112175d	se.dou.LcboFinder
5c20397ff602711630c23c6b3f3d528d	com.ccwilcox.bft
5d50f937b75d53aeb8187ee2c820edf5	net.jimblackler.quickcalendar
5e94ca83c40d783a5e5692a811bb7399	com.jjkim.gasprice.lite

Benign sample set, continued

MD5 hash	**Package name**
5f66009d9758ec5c87880ca939c0d7e9	com.bipgeeks.riverwallpapers
6040aabca376575c88465b05b5e46b5c	com.cyberactivities.guitarsquidfree
629a639dca4a7548587d5dcc78b9b2cd	com.goeswin.powerballusa
643d60e3bcf99652e9066b0015bd6411	net.leieuncretino.beta.hockey
667388681573c6c9a3a0cb819f6dbbe6	com.bai.GeoAlarm
66ede98974ecdbfdca1f2d29f4edd54a	com.polar.android.cbscgtaub
672b22efeb0503e673a9a4f3f05104d3	com.appspot.yongDriver_US
673d999791ab6f7aac73b8df33d8c1ff	com.art.star
674ac59a2449cc69bd2bec4e211ff0d3	com.appitise.shakerhills
687d6e671614530dae2643f2ea5ec9c1	com.aSoftDroid.turkey
692c1a4857bc36b36fd115ec0cbf3247	com.hs.app.millionairePotential
6a0c789ccc587bae6cd7303453850314	com.circlechart
6b198b3d161f35b02a2ecb518e40b78d	com.linewinner.free
6c27426db915511cba8aa770d3477f67	com.bh.android.MarketCommentsReader
6ca018f20d9c422eac91577f1440867f	com.allesapps.puzzlebox.sexyabs
6d9244025e13db939ea448177cf58d47	jp.mapp.yusha
6e5c860a2eac50e1cccb164d8c4dbaba	com.boa.megafartbutton
7017119b579e12991b4c6822fd06ff9d	com.timeflies2010.android.journalmap
7131366d608d3ca9d9c236e6d305385d	Sizep.yjh.pj
7253ad5262243a7fb9a1d16c6c9fcc9c	com.mixzing.basic
72c9dc947fa2b0be0ce9cd40111e06d8	com.lcb.AHouseBoatOnTheStyx
73ec5587bc27d2c01fdeb9173cb50ee0	com.tearn.kcarib
74c4204c8edb99683aa9d401b2f0635e	com.syborg.nathansnumbersfree
75bb0a4a8520f49f5422ddb992717227	com.dg.zeng
76a68d625eeb36017351da94c4b07161	com.oslwp.doraemon
76d507a7595b984677ef387654a3d809	info.androidx.workcalenf
7889a8f9d6544f64f6033aeca1648c26	com.fsellc.slidepuzzlepomeranian
79e1952fe498af968e2695806c7da8fa	com.appastrophe.comics.comics_people_wolverine
7a02f8877ad4e0bf5792441dd63df05b	de.radioland.mobile.radio.android
7a90582ec4dbd1069584285e93c86fe2	EnglishSongsTop100.ynot.com
7aa8e4ef50c0fc6ba2490551703dfcdc	com.axant.domenicali
7b4c48d21962774df434aea2d41f9060	com.omerfarukozdemir.mustafakemalataturk
7b64878d7757e06fa76756c72268886b	com.appulearn.musictrivia.android
7b949dcd5d61bcde2e2391e2cefc98b2	mkoss.android.biorythms
7bb2974d249c8bc0210270b04ef6b925	com.mrselected.bellyeraser.lifestyle
7cc24c499be65b980798638781a9ae02	com.zlatkoStamatov.quiz.vampireDiaries
7d242f03c64e17857fad2c6d8b451a50	com.bytesequencing.android.dominoes.free
7e3826872081c4799fc6f7cdc6adb7d1	com.kdrv.android.weather
7e4ae99fd204659d06319466e60a715e	com.jhjo.ringtone
7e7906b032d387d50da959567210a7f0	statsheet.statblogs.GopherBall
802d5eb74b3109778e04e847aa072735	com.appmakr.app221215
82d0ef403cb8138221739e4d59d8d06c	com.dreamtree.littlekittywallpapers
837fcdee26accf3684188c89db06433b	com.cellufun.launcher
849e9f7fc3e0a1ed06c93c0200c28d9c	com.rovio.ume.u1311578526281
85a28a91483f8e62bdc26521615c5737	com.pubcircapps.makeyour756
85e2021b0f806880c5f388a93e68a6ff	info.sabelan
86aec8695314ddec23cbca00ae12e6cf	com.bhsoft.expensesfree
8745a1a0922f50604b2f9b68ce5ad56d	de.sellfisch.jumping
8894d804bc859ffbb71e36c964855338	com.angryhippo.blip
897028ff3bd8ae4d00d1936953b4d012	com.nicripsia.otracking
89ef9998aede268ab154d1debd2ee869	com.linuxmobile.android.deeper
8bc342f2f238a65b0c26fa19e06f0dea	com.fantasticapps.travelguiderotirgumures
8c52799325bc1a297685945135851614	com.walmacapps.setasringtone
8c57e562f2ee250084b1c808cd370941	com.popapp.poppic.chun
8d0364d9eefd77b0b80a6ab12577a86e	net.notify.notifymdm
917770eccad83691a1db063128128215	com.victorvieux.android.convofy
917c8ef3111bb96cf75581a54ff8f93d	eu.reply.sytelgmbh.android.PlateAnalyzer
91d77a38ca06a307723b10c006473ee8	com.gmail.jaggersoftware.antitheftalarm
934e932f1b1d78620e350460effa072c	statsheet.statblogs.BusterBeat

Benign sample set, continued

MD5 hash	Package name
9529da32aa2e674a6bbec241239781ac	com.jayuins.movie.english.lite
9670c240cdcd1fb1bdb41afa9afaf6c6	flying.wallpaperlucky071207
9683fcdb2aa58e9496aee2e29e886aa2	com.backma386.puzzle
971ccdad781e34f246c7e49b27d90216	com.rcreations.amberalert
98d6c7a2413fc70fe5f928e4097b4225	com.blood.alcohol.level.calculator
9a4437cf405a034fda5f6b7c830ca993	statsheet.statblogs.HuskiesUpdate
9b67d404bf404f10b4db8cd1e95ce58a	com.crossfield.waterrocket
9d1855a20eb2baac27597b1e558d7038	com.jpn.bestyle.kamehameha
9d5a2c834d7355ded9085b0d1ffd6cc6	com.moram.rtkfree
a092c6d948772510d98de178f0d8e5e5	com.bhpro.soundbankhealingfree
a137e3fcbc73c472cb5b2de6ed097479	com.lsn.localnews291
a1deafd88ba4b7939038d6ffade11c8f	com.karmic.azenquotes
a38e5b2974acc72e77a866b9bb0e7bca	com.f2fgames.games.hungrybear.lite
a554aba36824595818d6274c29130275	com.explorationguides.android.waterloopinckneytrailmius
a712140cce2358543ebab8369408cc0e	com.kcc.mgic.android.game.spidercraze.demo
a76c68f67947635e7d737c516e427a2e	com.jtpub.hornsandsirens
a88aa41490b039c7897f286a2bd22088	statsheet.statblogs.ButlerBlueFever
a979e9f3c5104e488218428edb7fefc4	com.kasa0.android.slitherpuzzle
abd0a14a3c6c79863c006a4d82818425	com.appsbuilder33854
adabee8f4422341b479d125cf1b1eba4	swan.walljun20j
add02ba28c7eabe6485669520174fc16	net.androidresearch.xmasball
aeb8c37e12cc6621370fef1032ce42f1	com.busybits.games.stackit
af2d87b4fbfe8e7dad770f783bc6feb9	com.kennyrogers.droidradio
affedea957cba3d2768bb479c9499d61	com.appmakr.app158720
b07a5c4dd32503ec67fe452e9f17eab6	com.chs.headrate
b1272e0e80ae5ffe734aa06b953768ac	com.VocabularyTrainer_V2_L1_de_cn
b13acae6fe42993e24121db13c98379b	com.tophumour.toto
b437399c3b9c1d58d0c51a0e0b0c9b21	sp.app.bubbleFlood
b669a729090f7d1b4573ad1b2437b5e6	com.texterity.android.JPM
b961bae1bdb11af874e14744ee7a6ca1	net.mobabel.packetracerfree
ba0b08ea84abea560de30f1b5f0c5175	net.jjc1138.android.scrobbler
ba9a81c611d3aa10df6f793df9212174	me.scriptmatic.BHQADM
bcf53a95f085975bf4d6391a4211adef	jp.co.skynara.S11002632
bd0bd87e2d38528cd6a79c97211b7bbe	coldstream.android.nuclearlite
bfddeff4ce88a40090f86fca5d5efe87	com.mediafriends.chime
c003a20a7631711f311c95a13c410eba	com.aseanmobile.chinesedictionary
c0bf2b3f7a190182f3545ac250e494af	net.oxdb.CalcBMIs
c13a42f84aa4fe928f5b6931929b4cb0	com.alaskajim.movietrivia1960s
c16a469b79059a888e9a6eaea37de437	luck.of.wise.mushdoor
c19d6bb5616fe02557597680b696de5e	com.sequence9designs.recordscratch
c29b86d04f62b3f4bf2835c63520ad1e	com.crossfield.ninjabomber
c38d317aa0b6c76fdd44505b4113f2cf	com.millionairemateapplication
c4ef5fec276843aa505b76b8f7ef328c	com.iwpsoftware.android.picturegallery.strategicbombers.b52
c51c9d75dba47e8d19ae38cb7bab897b	statsheet.statblogs.AlcornNation
c720f1f72daa893e2f0b70d40ba145f2	com.diordnaapps.twlotto
c77265cc5dbf1e559fb0c6972755ca53	cyl.datinghoroscope
c7aef851672bbf77de139b5a469ab6ec	com.flagsibh.buyinglist
c7d030f728ce5623fce6ea68e18bdb82	com.Relmtech.Remote
c925ea1cc3672f28d6895c8971764105	com.narble.quotes.mathematician
cb7c0637c4c55a64ae51c1838dba2893	circle.wallpaperlucky041405
ce13e0500ac43ffae7c65f0cfa14eba9	cz.gibosms
d2894e15db13ea78546f61bea663cc4f	com.bzyg.xingqutanmi
d3f195c91b29a0fcb68078309a257271	jp.AppDevMan.K_TaiBrow
d6d927c725ec0ef8181bc97689e280c6	game.hon.cardfindonline
d704c2705474098478469f0c02da79e4	ca.webpanda.wood
d7e1dee2c9003b952869a6803e4b72b7	com.kh.hit.my.face
dab0197b25a07b35cb141067be1f8b46	yami.wallfeb23a
db8538065cb1b179ddf5e65636effed2	com.bitknights.dict.engfre.free
dbd24b729ad2932528c00dd923e159d8	com.twistbyte.chatlingofree

Benign sample set, continued

MD5 hash	Package name
dc2c23e3e29b29da0545384ef1b2ed10	cn.bluesky.fingerbasketball
dc3fdb446b00a023d7cefe99be2b3992	net.jp.onaka
de2d908a3c3cc9e9d72552ba5feeb113	com.softdyssee.lifestyle.proud_jobs_lifeguard
dff0fd6860a351b59152c056e6b4516d	com.fantasticapps.travelguideus.ca.glendale
e168c74f05ee6d6853a06eeb845f4a52	color.wallpaperlucky061609
e1fc0ddb8e6cb9385cca751586b6ff58	irdc.flower
e4bdfe954463bb2b1db1db01d5b48950	softkos.uc
e4f8437970b9ba9f5ff84fc572173522	com.mobiders.pagoda
e591201357c95d106ce6e4b616b8ea69	com.maxdroid.valentines
e5bb68cb9313f5b38fc83d86511ea64b	ks.packs.anotherDay
e751c54ec36d7854b2681c710b689f74	com.fantasticapps.travelguidehkhongkong
e970ba5d6ca4f1ede8e74843d93a45da	com.jasc2v8.abc.demo
ea7391cdb5445a92138a49eacd0bc674	com.klaymore.dailycomix
ead6626facc09d0de4fd84bd991f03f9	com.fingerSwimmer.www
eb496d283311d2ad009f56faf22b8181	com.richstern.scribbler
ebb8c145dba2ec052513a3f8be811df4	com.GirlsinDressestwo
ebf8a74e9549d14bf25de93c133fcc4e	com.kenagard.widoobiz.android
ec1c6b337fdd14fd3953b245a6a526da	com.bestwp.B4515
ece2b726e8c1e7c22c2630558507a28a	com.frontapps.fingerscan
f00034eba17c78bbc8525da9e5f4e88e	de.itcampus.mdr.android.mdrsachsenanhalt
f240abe83b8da844f5dfdaceba9a6f7e	com.AndPhone.game.Defense
f2c3afe177ef70720031f2fb0d0aa343	com.skylineapps.opentech
f40759b74eff6b09ae53a0dbcabc07d4	mango.walljun30m
f40b2e884a9952560b8af1675d1850c0	com.v1_4.B6CE724F0A201822.com
f578f8e4c244d206c8b67fedfe1841d7	com.appcookr.app_128
f5d6b6b019949329ef0de89aca6ac67e	com.baste.bender
f67458e82a7e9ecf51808083fc52f2ed	com.mobileagreements.club.extra
f6a0e9573810d3da8a292b49940b09e2	com.probaseballapps.sethsmith
f6a3e3ae9e071a28107952a5421132b7	com.sancron.ringtones.sb.funnysmssb
f7f02beff775d3a33e5299784b4f35ce	hu.hermeszsoft.origo.mobile.android
f81fbe1113db6ca4c25ec54ed2e04f42	com.hetverkeer.info
f8bdccce4f4462af87b358a8022efa27	com.whiz.android.weather
f9b5afdff92f1eb5c870cf4b601e8dc1	com.snoffleware.android.rationalcalcfree
f9bb1a7e1169e14381ad487351ce25a6	com.xrhome.amapp.smartphoto
fb891ea00a8758f573ce1b274f974634	height.wallfeb28m
fbefbe3884f5a2aa209bfc96e614f115	com.accesslane.screensaver.shootinggallery.lite
fd1af0690436028285a889c1928041ca	org.steele.david.silentOnOff
fe99a0177dc38b8f6707ce4f180ad079	com.bestwp.Ispring

Table A.2 enumerates the 242 MD5 hashes of the malicious sample set that was used in this thesis. For each sample, the corresponding malware family as it was detected by Kaspersky is listed (queried via VirusTotal[1]).

Table A.2: Malicious sample set

MD5 hash	Family
0018874837a567609e289661cd418639	Trojan-SMS.AndroidOS.Placms.a
003d668ef73eef4aaa54a0deb90715de	Backdoor.AndroidOS.GinMaster.a
0059a2d57f9bc3756652a3703c169ca4	Backdoor.AndroidOS.BaseBrid.bj
010982ffdc311f8a8236bde676cd6561	Backdoor.AndroidOS.KungFu.a
019d1fa6e7aaf3c13dede5f445507992	Backdoor.AndroidOS.GinMaster.a
0238a007a38221c13a4fddf7d3771314	Trojan-SMS.AndroidOS.FakeInst.a
02718a3a788e4e34f07b658aa284d680	Trojan.AndroidOS.FakeDoc.a
035548473f8b2b44b50301ea6000d11f	Trojan-SMS.AndroidOS.FakeInst.a

[1] http://www.virustotal.com

Malicous sample set, continued

MD5 hash	Family
036c0c18a99e425d3c189c4467016799	Trojan.AndroidOS.FakeDoc.a
046f32fd5db4097fd38647cb5d607206	Backdoor.AndroidOS.KungFu.a
06d6b20d0a0469ba793706fc2b272848	Backdoor.AndroidOS.Kmin.d
0985eb42014d865543d0c9d95d8a14e2	Trojan.AndroidOS.FakeDoc.a
0a09dfc1b6d3fdbbb5c02bee40054faa	Trojan-SMS.AndroidOS.FakeInst.a
0a5816e203b6b5a4f5479cd683729a97	Trojan-SMS.AndroidOS.FakeInst.a
0c41d066a2c9e10e71481f20cc60d4f0	Trojan-SMS.AndroidOS.Placms.a
0c8bd7f64a5b69a11a304c83941d0ea3	Backdoor.AndroidOS.Kmin.c
0cc2c871ec2a37b72098b02ca4392fe9	Trojan.AndroidOS.Gamex.a
0e632dd6c9c60898631a2723b0cfe958	Trojan-SMS.AndroidOS.Placms.a
0f61a048cace9d03fbb1dfe7390a6527	Backdoor.AndroidOS.BaseBrid.ae
12436ccaf406c2bf78cf6c419b027d82	Trojan-SMS.AndroidOS.FakeInst.ed
12830bbc4503cdadaf60becd20ba4fc5	Trojan-SMS.AndroidOS.Opfake.bo
128629e7a3fd7f28ecff2039b5fd8b62	Backdoor.AndroidOS.KungFu.a
1295a47e650e818d7fca5ebe181a0261	Trojan-SMS.AndroidOS.FakeInst.a
1638d5cfffc4bcb2deec3c20ef09b330	Backdoor.AndroidOS.GinMaster.a
17278c15c054f0802dbdb13f23965198	Backdoor.AndroidOS.GinMaster.a
176aeb66e7a9301ca0035abb91253ac6	Trojan-SMS.AndroidOS.Opfake.bo
17eebcde25d7ce816a2aa7ad8a0f8264	Backdoor.AndroidOS.Kmin.a
1c91795299300eecdc5c6d3b9ffc0b56	Backdoor.AndroidOS.Kmin.a
1d52dcd5c2099ef2664398419d154b62	Backdoor.AndroidOS.BaseBrid.ae
1e8a6884470d5496ebd7e6eed902fa91	Backdoor.AndroidOS.GinMaster.a
2022bbab6d6a89ff7c923ea54bd49ee5	Backdoor.AndroidOS.Kmin.c
227a4c675f2384da13af381938e432df	Trojan.AndroidOS.Gamex.a
228eaea19c08f0912805669e081da02f	Trojan-SMS.AndroidOS.Placms.a
22b79389e097853f6fc573e7ffbe6c04	Trojan.AndroidOS.FakeDoc.a
230ee6ed41f47efd862b3d66ee8f42f4	Backdoor.AndroidOS.GinMaster.a
2573c404acf459c1e11f5124c1a75073	Backdoor.AndroidOS.KungFu.a
2609905341475941eef9dce106609cb6	Trojan-SMS.AndroidOS.FakeInst.a
263e0a743cb3729096cceccea6caf58f	Backdoor.AndroidOS.Kmin.a
266cfb7cb1fd1cea802258de2a011049	Trojan-SMS.AndroidOS.FakeInst.a
26a5328596d96b08627c1afb408acf8a	Backdoor.AndroidOS.GinMaster.a
29ea206bd85a835dead88443fbb1cf1f	Trojan-SMS.AndroidOS.FakeInst.a
2bc92bf1bbfadbc928d49fdc70b3035e	Backdoor.AndroidOS.BaseBrid.bj
2c53e182ed5669f798402e638bbd02cc	Trojan-SMS.AndroidOS.Opfake.bo
2dba8e5f96961e0fdf584e31d1bb8bc9	Trojan-SMS.AndroidOS.FakeInst.a
2f2cd901375d064a9b3e7734789c77fb	Trojan-SMS.AndroidOS.FakeInst.a
2f80ed362d04fc249e619b5f56fb5b0d	Backdoor.AndroidOS.GinMaster.a
30908fecdc8d811fc9e94280a1648bd0	Backdoor.AndroidOS.KungFu.hb
30a579fb2a39dab5c8fe82c9a9a6383d	Trojan-SMS.AndroidOS.FakeInst.a
32ff30b47a183c86a83840f7028dea00	Backdoor.AndroidOS.GinMaster.a
33094d472a736f4a6706de8f8db71804	Trojan-SMS.AndroidOS.Opfake.a
37eea1dc578ab0efaee56dce13d3d84c	Backdoor.AndroidOS.Kmin.e
37f894dff34637d4256dc0ebdc645e70	Trojan-SMS.AndroidOS.Opfake.a
38a33a960821e8980f9b60b9c6662f79	Trojan.AndroidOS.FakeDoc.a
3913db14a3237950e3ae858cae5dda75	Backdoor.AndroidOS.Kmin.a
39f874b984ee34310b126d543fed3706	Backdoor.AndroidOS.GinMaster.a
3bd9542cd86fcf966a1bd6a41389e7d9	Backdoor.AndroidOS.GinMaster.a
3ce1442b9f4ddb5d0bef33d2b836cf7a	Trojan-SMS.AndroidOS.FakeInst.a
3cec22da373dad044c9fd40c87689057	Trojan-SMS.AndroidOS.Opfake.bo
3dc29d1a5d7b5e0df3bbf12190fb62cc	Backdoor.AndroidOS.BaseBrid.cr
3e018fd52ed643b90f12947801d85cd3	Backdoor.AndroidOS.GinMaster.a
4005dd74246c5e97a0a0dd860d29bee5	Backdoor.AndroidOS.BaseBrid.g
432ab46f7a77b10a1ab1b0d476f52bcc	Trojan.AndroidOS.Gamex.a
4446e1f537f80b11de4ef19893dc0463	Trojan-SMS.AndroidOS.FakeInst.a
44dcdba726344eb52fb31f6ec41df8bb	Trojan-SMS.AndroidOS.FakeInst.a
4790a6ba7717306876216aa28a4492f7	Trojan-SMS.AndroidOS.FakeInst.a
49ad664ed30b51c000137adf6c415d9a	Trojan-SMS.AndroidOS.FakeInst.a
4bd002a2e2c5d8772ce4fd2c095da171	Trojan-SMS.AndroidOS.Placms.a

Malicous sample set, continued

MD5 hash	Family
503595a663c3cb776b432fe33cb35280	Backdoor.AndroidOS.Kmin.f
528b7de40cf0eb6ef477a7f38c57d4d8	Backdoor.AndroidOS.GinMaster.a
53f5696da3fee6db894aed6fae720ba5	Trojan.AndroidOS.Gamex.a
55c3ef423e9c8075001a98f34e5c548b	Trojan-SMS.AndroidOS.FakeInst.a
566240dd12bba783d49d6ba7e463081b	Trojan.AndroidOS.FakeDoc.a
566787a4656ad47606c3c90b66e0f850	Trojan-SMS.AndroidOS.FakeInst.a
568d67831947be03262bc7486dbeb140	Backdoor.AndroidOS.KungFu.hb
568dd1845ef09f34f8278eb7c6c4f80f	Trojan-SMS.AndroidOS.Placms.a
56b7fff85e7f1ae0d1727a2beb093b77	Backdoor.AndroidOS.BaseBrid.bn
56dfef2e97970609968bf9556114f8b3	Backdoor.AndroidOS.GinMaster.a
585bd45a03ac050d9cceb8c2032cdd72	Trojan.AndroidOS.Gamex.a
5a3c92f9b9c6eeac3b1151030339ec54	Trojan.AndroidOS.FakeDoc.a
5b6f15437c0627bf44cf346a139dc027	Backdoor.AndroidOS.Kmin.f
5f47cfd0fda265168462007c9e50b456	Trojan.AndroidOS.FakeDoc.a
6018fbf0ec36618da323a0b41fac02e4	Trojan.AndroidOS.FakeDoc.a
6559e92ac9b7c4209e22af2f628e6532	Trojan-SMS.AndroidOS.FakeInst.a
65f44d56a0676a2fdbc39180bba42c03	Backdoor.AndroidOS.KungFu.hb
660780b794ec35dcd587f6437e04caf0	Trojan.AndroidOS.FakeDoc.a
6724bc8a45de7bc489433fd6c928b37e	Trojan-SMS.AndroidOS.Opfake.bo
676e19af453394bea375b85c95ddd9ad	Backdoor.AndroidOS.GinMaster.a
68d60da4ca19572d58de275a1a77c9f1	Trojan.AndroidOS.Gamex.a
6b4cc83e23ac611c75dacc0b4cd698bd	Trojan-SMS.AndroidOS.FakeInst.a
6f501f23d11f38a99a8da644d078fcb7	Backdoor.AndroidOS.KungFu.a
6fb35cbcbe6a6bcab53c60dbff35d876	Backdoor.AndroidOS.Kmin.a
714e9fde06f7d6805ccc4d303b00280e	Trojan.AndroidOS.Gamex.a
73ad43a68890220c030623b58cf42d8e	Backdoor.AndroidOS.GinMaster.a
74cadaeeb296d96347f71a0aa827cc40	Backdoor.AndroidOS.Kmin.e
758e6a40de51292611be7d5323f74088	Trojan.AndroidOS.FakeDoc.a
75bec5da538e24f2b63f722a9b321bd8	Backdoor.AndroidOS.KungFu.ki
7639883e747b440951b7f1d72525fa17	Trojan-SMS.AndroidOS.Opfake.bo
77684a8f14e5859b6734124447e83c38	Trojan-SMS.AndroidOS.Opfake.bo
79b4d8e85755bf4cee0f92c2ca0d3c57	Trojan-SMS.AndroidOS.Opfake.bo
79e40d747b90137f5ab8dd17047d2679	Backdoor.AndroidOS.BaseBrid.bj
7b6d92c16407ada04ef023f9d8f9004e	Backdoor.AndroidOS.GinMaster.a
7bd81da3c4a01f2ada043879ed5ce059	Trojan-SMS.AndroidOS.Opfake.bo
7d5213edab1f71f41983b1b88cd4f683	Backdoor.AndroidOS.BaseBrid.bj
7d8930d68f9409d29680e0eb4bc5c822	Trojan-SMS.AndroidOS.FakeInst.a
7d8be7bbc4597d1302ce4eaf34c9f579	Trojan-SMS.AndroidOS.Placms.a
7e4959e409e277d564261c18e8604747	Trojan-SMS.AndroidOS.FakeInst.a
7ee5f03b09cc71116f472d9f9a28247a	Backdoor.AndroidOS.BaseBrid.bn
7fda2f90fbc45e7c165a820772e4e42e	Backdoor.AndroidOS.Kmin.e
80279e8517876a47a07bad63427b6eed	Trojan-SMS.AndroidOS.Opfake.a
8223350420eb11293415809cc5e82c5c	Backdoor.AndroidOS.KungFu.in
83a5bb4cbab99da0676b8cc80dbe53c7	Backdoor.AndroidOS.KungFu.ki
847ecfbc03ccb0417550abf22739f988	Trojan-Banker.AndroidOS.Zitmo.a
85bc8ed574554393e7e11ec42df89129	Backdoor.AndroidOS.GinMaster.a
871fc929fd8e066bbb3badadb4b6321c	Backdoor.AndroidOS.KungFu.a
89b0fcae589b3a7d5e79fa870e0f47ce	Backdoor.AndroidOS.GinMaster.a
8aac3c2bc718701749a1d485000775c8	Backdoor.AndroidOS.Kmin.a
8ae9cc9e53baeab4c5ea9f4e79091502	Trojan-SMS.AndroidOS.FakeInst.a
8b488822a33ae2f6b316d90e92fb5872	Trojan-SMS.AndroidOS.Opfake.bo
8b5132504377078d4a7281b45f9fae29	Trojan.AndroidOS.Gamex.a
8f7f191ab891059c6d55c33af69abbb7	Trojan-SMS.AndroidOS.FakeInst.a
9210c0c1aa3eb4de11d16a0b7072d94b	Backdoor.AndroidOS.BaseBrid.bj
9234e1fe084d21a84fb56028c1aafe9c	Backdoor.AndroidOS.KungFu.ht
923f00829bd4fb49e85d6688d54fd45c	Trojan-Banker.AndroidOS.Zitmo.a
9300841786cd98af1921fc41b0f5e1aa	Backdoor.AndroidOS.Kmin.c
936162a5cdfc1e73a1d8740ab1d164b2	Backdoor.AndroidOS.BaseBrid.ae
93fd1e8021fb3cd7c3a6ecf9135baef1	Trojan-SMS.AndroidOS.Opfake.bo

Malicous sample set, continued

MD5 hash	Family
959aa8b2e31bfe6429b92863f9630591	Backdoor.AndroidOS.GinMaster.a
95f08c6f15406f43dbacb744c27ae72d	Trojan.AndroidOS.FakeDoc.a
96c7d4e21d8fb03f94b703c43233933e	Trojan-SMS.AndroidOS.FakeInst.a
9954b925437e68ce28946b941e59cc6a	Backdoor.AndroidOS.GinMaster.a
9a40f64e91443c2a3b2d64902efb9e66	Trojan-SMS.AndroidOS.Opfake.bo
9b38a69982ae2bf061d59f154bad393d	Trojan.AndroidOS.FakeDoc.a
9be24f5e7c5f9faaaba0aec59e5e1982	Trojan.AndroidOS.FakeDoc.a
9cfd82495b917dfabe3c72eabec213a6	Trojan-SMS.AndroidOS.Opfake.bo
9d62dda0b7b1ae7795dc6dd701508765	Trojan.AndroidOS.FakeDoc.a
9d7a1ca92904302db105c2c755f57586	Backdoor.AndroidOS.KungFu.a
9e5affc92dd32dbf11ac1b80941ca21d	Trojan-SMS.AndroidOS.FakeInst.a
9f37947e358d9d3f1ea15d86d82695da	Trojan-SMS.AndroidOS.Opfake.bo
9f4612ebe8f5f8ceee02af56d971e3e2	Trojan-SMS.AndroidOS.FakeInst.a
9f857bb83c4237e54adabd23cafcd332	Trojan-SMS.AndroidOS.Opfake.bo
a1c1a7ee09030fc0432f399401500480	Backdoor.AndroidOS.GinMaster.a
a1c57ec4a8823549db0cea2962cf934b	Trojan-SMS.AndroidOS.FakeInst.a
a373823c2a2df566c28be35892efec60	Trojan-SMS.AndroidOS.FakeInst.a
a4217a8256b8b5858ba8b16ffc386ce1	Backdoor.AndroidOS.GinMaster.a
a6b7d5652760aec0575f6971d3dc8659	Trojan-SMS.AndroidOS.FakeInst.ed
a73a185c19d97bb42ec3e4edb375a2ed	Backdoor.AndroidOS.GinMaster.a
a74b1d46ed083695be69586c69c0f81c	Trojan-SMS.AndroidOS.Placms.a
a84b882ecd9d51c39afcdb870bb0aaee	Backdoor.AndroidOS.GinMaster.a
aaead6e17631216f24a0b885b0aca7a5	Trojan-SMS.AndroidOS.Placms.a
abd5dd4db04e228fc504a62f81ebbde6	Backdoor.AndroidOS.KungFu.hb
ae1bbad09466168414e9bbc65f37033a	Trojan-SMS.AndroidOS.FakeInst.a
aecd04d69795f6f382ffce0ff85f7fb6	Trojan-SMS.AndroidOS.Opfake.bo
afd7b5d39c555f4eb73cffa5f4f623e6	Trojan-SMS.AndroidOS.FakeInst.a
b001c3e8b7583d2cc1ee26e0b53296a6	Trojan-SMS.AndroidOS.Opfake.bo
b09009ae2da2bda6dc0857e6308a6fbe	Trojan-SMS.AndroidOS.Jifake.d
b289df7defcef4d58162d77fa5d362d8	Backdoor.AndroidOS.Kmin.f
b4a07c8a8832586187ed7a3faf668856	Trojan-SMS.AndroidOS.Placms.a
b5d0e736412d1b32d9a5c01b973ed2b6	Trojan-SMS.AndroidOS.Opfake.bo
b5f9db56b068aad3b12e79bfc1ff0bab	Trojan-SMS.AndroidOS.Placms.a
ba1d7ad4f2bd528384a07ab2b2f0d67b	Trojan-SMS.AndroidOS.Opfake.bo
bad34491dddf683b68093e04dc354e5e	Trojan-SMS.AndroidOS.FakeInst.a
bba424b89b515da635a3ff1509609fee	Backdoor.AndroidOS.GinMaster.a
bc7def5a3b3cba38710dbbc7ef865de6	Backdoor.AndroidOS.BaseBrid.bj
bd2403966eba9e95b9479640cfdecb94	Trojan-SMS.AndroidOS.Placms.a
bd70ebebcb5749f8b7ba3f52b9321e79	Backdoor.AndroidOS.KungFu.hb
be4f6b944700485a90c7743d6b7f99bf	Trojan-SMS.AndroidOS.Opfake.bo
be9ca2f1c159f192f5fb67c964c5c724	Backdoor.AndroidOS.GinMaster.a
bf17b70c385cd70e1f3016e72f97242b	Backdoor.AndroidOS.KungFu.a
bf4e03e02829b95a20680c4903f49807	Trojan-SMS.AndroidOS.Opfake.bo
c01a1ca0ea4aa4be9d56c876da97208d	Trojan-SMS.AndroidOS.FakeInst.a
c16b7422ef5fdf06cea68f0d7d5e471d	Trojan.AndroidOS.FakeDoc.c
c3aa6c02a7ba986a6522fe31bad6f00f	Backdoor.AndroidOS.GinMaster.a
c3b65bec97c87a2e99804d35b65eb182	Trojan-SMS.AndroidOS.FakeInst.a
c4573c02744ef224deadbba46292f795	Trojan-SMS.AndroidOS.FakeInst.a
c4f9575adb31940b5c990ca829c607ab	Backdoor.AndroidOS.GinMaster.a
c5f6d5289097dd236db0a131af6ffede	Trojan-SMS.AndroidOS.FakeInst.a
c6566c1b22ee500ab91d971e99cd7dd5	Trojan-SMS.AndroidOS.FakeInst.a
c6d940417473b10bf65bd2bda804b3fd	Backdoor.AndroidOS.GinMaster.a
c71135730b5e26e4c148e941a446bf79	Trojan-SMS.AndroidOS.Opfake.bo
c7113d3acf769a58c1f21f8534d29f8f	Trojan-SMS.AndroidOS.Opfake.bo
c7b53bf5bd28af3ad84bf53dbb94bfbb	Trojan-SMS.AndroidOS.Opfake.bo
c99ba1121c6c72d851d937a0fa59aea2	Trojan-SMS.AndroidOS.Opfake.bo
c9e3af6a4429197c05c18408f9f287ee	Trojan-Spy.AndroidOS.Zitmo.b
cb04487016cc6dbf4481c2399a4b3b78	Backdoor.AndroidOS.BaseBrid.bn
cbfa867dcfa8cdfab76dee5393113a6d	Backdoor.AndroidOS.GinMaster.a

Malicous sample set, continued

MD5 hash	**Family**
cdc7b23acd9f66ca68d66c4010b49964	Trojan.AndroidOS.Gamex.a
cdeb94762c8975617d0a883d2f1428d9	Backdoor.AndroidOS.BaseBrid.ae
ce4eaaf64a35f9be4a4e5f9c30ad3224	Trojan-SMS.AndroidOS.Placms.a
ceda0c9175b1cff39caf783627befcb1	Backdoor.AndroidOS.KungFu.de
cfad9d5cd164100080aee005eff08d14	Backdoor.AndroidOS.KungFu.a
cff20c72cfa0db47b7158f9b3f4d2a08	Backdoor.AndroidOS.GinMaster.a
d0e8dd0d51fe92e04c49a1a2d3b1ece7	Backdoor.AndroidOS.GinMaster.a
d1610b2ccd89bf7aea4a9e16b885f08a	Backdoor.AndroidOS.GinMaster.a
d2f410b63ef063f2bd204a4ed58b63a8	Backdoor.AndroidOS.Kmin.f
d3148c65bcb55031b572e0035570a248	Trojan-SMS.AndroidOS.FakeInst.a
d3cfbf352739d475f60ace44b4686528	Backdoor.AndroidOS.BaseBrid.cr
d41c6c23c6c80d76d705125dfe33e1b1	Trojan-SMS.AndroidOS.Opfake.a
d4493ff3e5d8d1cdae8cc4cf72397905	Trojan-SMS.AndroidOS.FakeInst.a
d78ac9eb24de72120664b6dff4c002a5	Trojan-SMS.AndroidOS.FakeInst.a
d97e71f5bd865fdad16f1f19d757197f	Trojan.AndroidOS.FakeDoc.a
d9b29aca1ab46c3fb136f55053d29a2e	Backdoor.AndroidOS.Kmin.d
d9cb8085f9dfd250fdcf669496c7e61f	Trojan-SMS.AndroidOS.FakeInst.ed
da71c18f32c63b10b2caea2b718067ed	Trojan-SMS.AndroidOS.Opfake.bo
dac16af9e6d007ee7905871f10c727a8	Backdoor.AndroidOS.BaseBrid.ae
db4989689102a69ce8216f208fdbe38e	Trojan-SMS.AndroidOS.Opfake.bo
dbbb45a2286e24d4dbf6e23ebc158691	Backdoor.AndroidOS.GinMaster.a
dc0c80fc5a5bc6cc816136ff7a8930bb	Backdoor.AndroidOS.GinMaster.a
dec8a6bec19e206c3a0303ed9c0c90aa	Trojan-SMS.AndroidOS.Placms.a
df10c7840e45ce5add284c66bb57308d	Trojan-SMS.AndroidOS.FakeInst.a
dfbc2814fb096ee0ce18a64d208017bc	Backdoor.AndroidOS.KungFu.a
e25db0b7d60e50b6fef48582a352b8c5	Trojan-SMS.AndroidOS.FakeInst.a
e32f9c0b948e251b48217438b1d2295f	Backdoor.AndroidOS.GinMaster.a
e40e0e2b3131430383e9bc34b376ff87	Trojan-SMS.AndroidOS.FakeInst.a
e50722973ed0a6c53b0613c07c92e983	Backdoor.AndroidOS.KungFu.a
e7db3851a49e3e099ed24e9f886128e0	Backdoor.AndroidOS.Kmin.b
e830a4982acbf6cc383d00e8e482753c	Trojan.AndroidOS.FakeDoc.a
e8fb65a64577a52b19ba329c217dd823	Trojan-SMS.AndroidOS.Placms.a
e94af95ca1ff5e9ef187eb335ebecd87	Trojan.AndroidOS.Gamex.a
ebb2fa0fd4c7116c589832e6a4140ef4	Backdoor.AndroidOS.GinMaster.a
ed06d911347858a506ff1c32a6b4c567	Backdoor.AndroidOS.BaseBrid.a
ed78482b420d07ddf8eb5b059fd57199	Trojan-SMS.AndroidOS.FakeInst.a
ede142e2a8273929b8d1a9a0d57c0ab0	Backdoor.AndroidOS.KungFu.ki
ee3b5ddbfcb866141e9c123b62695e1b	Backdoor.AndroidOS.Kmin.c
ef2e6880021c5c2909f9a61091b0ee47	Trojan.AndroidOS.FakeDoc.a
f181409e206cbe2a06066b79f1a39022	Trojan.AndroidOS.Gamex.a
f2f2709fc2c8961a7dcf0e167f73e353	Backdoor.AndroidOS.Kmin.d
f3194dee0dc6e8c245dc94c5435750a5	Trojan-SMS.AndroidOS.Placms.a
f342d8f0c18410e582441b19de8dd5bb	Trojan-SMS.AndroidOS.Placms.a
f42a7cdc8a7b65211ce0ca5610616596	Trojan-SMS.AndroidOS.Opfake.bo
f458ca5d41347a69c1c8dc99812485ee	Backdoor.AndroidOS.GinMaster.a
f46f75e4eb294d5f92c0977c48f5af4f	Backdoor.AndroidOS.GinMaster.a
f4d80df6710b3848bf8c78c1b13fe3b5	Trojan-SMS.AndroidOS.FakeInst.a
f55a7ad2ab8b3ac2447964614493fffe	Trojan-SMS.AndroidOS.FakeInst.a
f7ad9e256725dd6c3cab06c1ab46fcc2	Trojan.AndroidOS.Gamex.a
f98ae3c49ce8d4d5ec70f45f06601629	Trojan-SMS.AndroidOS.FakeInst.a
fd225d8afd58cdec5f0c9b0f7fd77f58	Backdoor.AndroidOS.BaseBrid.a
fd48609ba4ee42f05434de0a800929ad	Trojan-SMS.AndroidOS.FakeInst.a
fdbce10ece29f14adfb7ebe99931d978	Trojan-SMS.AndroidOS.Opfake.bo
fe3cb50833c74c60708e4e385bb8b4fc	Trojan-SMS.AndroidOS.Placms.a
fea4a07813c0c557b3d745111a27d124	Trojan-SMS.AndroidOS.Opfake.bo
fead2a981fc24a2f9dd16629d43a6969	Trojan-SMS.AndroidOS.Opfake.bo
ffbee6719c51d3a1c4a1717c0b00f8f1	Backdoor.AndroidOS.Kmin.f

Appendix B

Availability of Related Work

Table B.1 depicts the available research frameworks from Chapter 6 and at which URL they are available for download.

Table B.1: Availability of research frameworks

Framework	URL
Androguard	http://code.google.com/p/androguard
APKinspector	http://github.com/honeynet/apkinspector
apktool	http://code.google.com/p/android-apktool
Dedexer	http://dedexer.sourceforge.net
Dexter	http://dexter.dexlabs.org
Dare	http://siis.cse.psu.edu/dare
ded	http://siis.cse.psu.edu/ded
dex2jar	http://code.google.com/p/dex2jar
JEB	http://www.android-decompiler.com
smali	http://code.google.com/p/smali
Radare2	http://radare.org
Julia	http://www.juliasoft.com
Andrubis	http://anubis.iseclab.org
AppsPlayground	http://dod.cs.northwestern.edu/plg
CopperDroid	http://copperdroid.isg.rhul.ac.uk
DroidBox	http://code.google.com/p/droidbox
DroidScope	http://code.google.com/p/decaf-platform
Mobile-Sandbox	http://mobilesandbox.org
SandDroid	http://sanddroid.xjtu.edu.cn
ForeSafe	http://www.foresafe.com/scan
JoeSecurity	http://www.apk-analyzer.net
Andromaly	http://andromaly.wordpress.com
Apex	http://github.com/recluze/apex-core
AppFence	http://appfence.com
Aurasium	http://www.aurasium.com
Kirin	http://siis.cse.psu.edu/tools.html
MockDroid	http://www.cl.cam.ac.uk/research/dtg/android/mock
TaintDroid	http://appanalysis.org
SCanDroid	http://github.com/SCanDroid/SCanDroid
AndroidRipper	http://wpage.unina.it/ptramont/GUIRipperWiki.htm
PScout	http://pscout.csl.toronto.edu