Int. J. Inf. Secur.
DOI 10.1007/s10207-014-0250-0

SPECIAL ISSUE PAPER

Mobile-Sandbox: combining static and dynamic analysis with machine-learning techniques

Michael Spreitzenbarth · Thomas Schreck · Florian Echtler · Daniel Arp · Johannes Hoffmann

Abstract Smartphones in general and Android in particular are increasingly shifting into the focus of cyber criminals. For understanding the threat to security and privacy, it is important for security researchers to analyze malicious software written for these systems. The exploding number of Android malware calls for automation in the analysis. In this paper, we present *Mobile-Sandbox*, a system designed to automatically analyze Android applications in novel ways: First, it combines static and dynamic analysis, i.e., results of static analysis are used to guide dynamic analysis and extend coverage of executed code. Additionally, it uses specific techniques to log calls to native (i.e., "non-Java") APIs, and last but not least it combines these results with machine-learning techniques to cluster the analyzed samples into benign and malicious ones. We evaluated the system on more than 69,000 applications from Asian third-party mobile markets and found that about 21 % of them actually use native calls in their code.

M. Spreitzenbarth (✉) · T. Schreck
University of Erlangen-Nuremberg, Erlangen, Germany
e-mail: michael.spreitzenbarth@cs.fau.de

T. Schreck
e-mail: thomas.schreck@cs.fau.de

F. Echtler
University of Regensburg, Regensburg, Germany
e-mail: florian.echtler@ur.de

D. Arp
University of Goettingen, Göttingen, Germany
e-mail: daniel.arp@informatik.uni-goettingen.de

J. Hoffmann
Ruhr-University Bochum, Bochum, Germany
e-mail: johannes.hoffmann@rub.de

Keywords Android · Malware · Automated analysis · Machine learning

1 Introduction

1.1 Android (malware) on the rise

In recent years, smartphone sales had tremendously increased. This explosive growth has drawn the attention of cyber criminals who try to trick the user into installing malicious software on the device. Google's smartphone platform Android is the most popular operating system and recently overtook Symbian- and iOS-based installations.

But attackers are misusing this openness to spread malicious applications through common Android application markets. In previous work [30,31], we analyzed about 6.100 malicious applications and clustered them into 51 malware families with the help of the *VirusTotal API* [20]. Nearly 57 % of our analyzed malware families tried to steal personal information from the smartphone such as address book entries, the IMEI, or GPS coordinates. Additionally, 45 % of our analyzed malware families sent SMS messages. Most common was sending these messages to premium-rated numbers to make money immediately. The last main feature that was implemented in nearly 20 % of the malware families was the ability to connect to a remote server in order to receive and execute commands; this behavior is typical for a botnet. Another detailed and well-readable overview of all these existent malware families is provided by Zhou et al. [37].

1.2 The need for automated analysis

Given the enormous growth and amount of Android malware, security researchers and vendors must analyze more and more applications (apps) in a given period of time to understand the purpose of the software and to develop countermeasures. Until recently, analysis was done manually by using tools like decompilers and debuggers. This process is very time-consuming and error-prone depending on the skill set of the analyst. Therefore, tools for automatic analysis of apps were developed.

The classical approach to automated analysis of suspicious applications is *static* analysis. Static analysis investigates software properties that can be investigated by inspecting the downloaded app and its source code only. Signature-based detection of apps, the common approach by antivirus technologies, is an example of static analysis. In practice, malware uses obfuscation techniques to make static analysis harder. A particular form of obfuscation used by Android apps is to hide system activities by calling functions outside the Dalvik/Java runtime library, i.e., in *native* libraries written in C/C++ or other programming languages.

In contrast to static analysis, *dynamic* analysis does not inspect the source code, but rather executes it within a controlled environment, often called sandbox. By monitoring and logging every relevant operation of the execution (such as sending SMS messages, reading data from storage, and connecting to remote servers), an analysis report is automatically generated for the analysis. Dynamic analysis can combat obfuscation techniques rather well, but can be circumvented by runtime detection methods. Therefore, it usually makes sense to combine static and dynamic analysis, which can be done in many different ways (one example of such a combination is shown in Sect. 3).

1.3 Existing Android analysis systems

Similar to the development in the desktop PC world, the early systems for analysis of Android apps used a static approach. A typical system for this approach was proposed by Schmidt et al. [28]. They attempt to extract the function calls from an Android application (using the *readelf* utility) and compare the resulting list with the data of known malware.

Another example for the static approach is *Androguard* by Desnos et al. [9,10], which decompiles the application and applies signature-based malware detection. This system is completely open source.

In response to static analysis systems in the desktop PC world, malware authors developed various obfuscation techniques that have been shown to be effective against static analysis [22,34]. This is also an emerging trend in Android applications, and it is clear that static analysis alone cannot ensure complete analysis coverage anymore. Therefore, researchers have begun to develop systems for dynamic analysis of Andoid apps.

One of the first such systems is *TaintDroid* by Enck et al. [12]. It is an efficient and dynamic taint-tracking system that provides real-time analysis by leveraging Android's execution environment. This system was complemented with a fully automated user emulation and reporting system by Lantz [21] and is available under the name *Droidbox*. Droidbox is an effective tool to analyze Android apps; however, it lacks support to track native API calls. In fact, we are unaware of any system that supports native API call tracking during dynamic analysis to date. *TaintDroid* and *Droidbox* are open source and publicly available.

Another interesting system using dynamic analysis is *pBMDS* by Xie et al. [36]. It uses machine learning to create user and system profiles for a specified behavior. Afterward, it tries to correlate user inputs with system calls by comparing their behavior profiles to detect anomalous application activities. This system was built for Symbian OS and tested with a very small dataset. *Crowdroid*, by Burguera [6] uses a similar approach, but with a much wider set of behavior data and with a more advanced monitoring system. CrowDroid uses `strace`, a debugging utility for Linux and some other Unix-like systems, to monitor every system call and the signals it receives. Crowdroid, however, does not consider information from Android's Dalvik VM.

The system *AASandbox* of Bläsing et al. [5] was the first system combining static and dynamic analysis in a very basic way for the Android platform. Unfortunately, AASandbox does not seem to be maintained anymore. Another system combining static and dynamic analysis is *DroidRanger* by Zhou et al. [38]. DroidRanger implements a combination of permission-based behavioral footprinting to detect samples of already known malware families and a heuristic-based filtering scheme to detect unknown malicious families. With this approach, they were able to detect 32 malicious samples inside the official Android Market in June 2011. Within their dynamic part, they use a kernel module to log only system calls used by known Android exploits or malware.

The system that is most similar to our approach is *Andrubis* from the Vienna University of Technology [33]. In their approach, they also use Droidbox and TaintDroid for automated analysis, but they are limited to applications that need a minimum of Android 2.3 to be able to run on a device (which is equal to API level 9 as can be seen in the platform versions overview of Android [3]). In contrast, we are able to analyze applications that support a minimum API level of 13 and will be able to analyze applications that are build for API level 17 and beneath by end of March 2014. This difference can be very important when you compare the market share of API level 7 and below (1.3 %) to the share of API level 17 and below (75.4 %). Additionally, they are not able to track calls inside native code at the time of this writing.

1.4 Contribution: Mobile-Sandbox

Overall, there are only few analysis systems that combine static and dynamic analysis and none that dynamically monitor both actions within the Dalvik VM and outside it in native libraries. Moreover, many of these systems are not readily available for research or not maintained anymore. In this paper, we seek to fill this gap by introducing *Mobile-Sandbox*, a system that

1. uses a novel combination of static and dynamic analysis as well as machine-learning techniques,
2. can track native API calls, and
3. is easily accessible for everyone through a web interface [8].

Within the static analysis part, we analyze the application with various modules to get an overview of the application. First, we perform several anti-virus scans using the VirusTotal service [20], secondly parse the manifest file, and finally, we decompile the application to better identify suspicious code.

Within the dynamic analysis, we execute the application in an emulator and log *every* operation of the application, i.e., we log both the actions executed in the Java Virtual Machine *Dalvik* and actions executed in native libraries, which may be bundled with the application.

The information collected during static analysis is used to classify the application as malicious or benign using machine-learning techniques. For this purpose, a classification model has been learned using a linear *Support Vector Machines* [7] as described in Sect. 3.4.

To the best of our knowledge, Mobile-Sandbox is the first Android analysis framework that has this capability.

For evaluating our system, we collected 69,223 apps from the most important Asian markets (like 92Apk, Anzhi, HiApk, etc.) and 6,162 malicious samples from different malware families. We then used the Mobile-Sandbox system to automatically analyze 10,500 randomly chosen apps from both sample sets. Within these 10,500 samples, our system detected 726 malicious applications and additionally 2 suspicious samples that hide their malicious action inside native code. Considering the fact that of these 10,500 apps only 500 had been classified as malicious by tools of the anti-virus companies, these analysis systems are overlooking important potential threats.

1.5 Roadmap

The remainder of this paper is organized as follows: Sect. 2 characterizes the current threat landscape in mobile devices especially for malware on the Android platform and gives some background on the Android platform. In Sect. 3, we illustrate our framework and explain the main ideas behind our static and dynamic analysis as well as our machine-learning techniques. In Sect. 4, we present the results of our evaluation. We conclude in Sect. 5 with a brief discussion and outlook on future work.

2 Background

2.1 The Android threat landscape

With the increase in smartphone usage and the distribution model of applications, criminals identified smartphones as a potential target for malware to steal private information, misuse it for premium SMS services, or try to manipulate necessary banking information (mTAN) on these devices. Very often, you can even find a combination of these threats within one malicious app. In this section, we give a short overview about current mobile threats and describe why the Android platform is the most targeted mobile platform.

Mobile threats can be categorized into two classes: *web-based* and *application-based* threats. The web-based threats on mobile devices are a growing attack vector used by criminals. These threats rely on the enormous usage of mobile browsers and their feature-rich implementations. Modern web browsers support features like embedded video players or support for video calls. Due to the nature of these features, e.g., parsing huge amounts of external data, the possibility for the existence of exploitable vulnerabilities is high. Additionally, attackers are able to trick the user to follow a web link, sent to them via email or social media, and infect the smartphone by exploiting a browser vulnerability.

The other type of mobile threats are application-based threats posed by third-party applications in the mobile markets. To install applications on the smartphones, the hardware vendors (like Lenovo, Samsung, etc.) created so-called *mobile markets* like Apple's "App Store" and Google's "Google Play". On iOS-based devices, software can be obtained from the App Store only. Furthermore, Apple evaluates every software uploaded to the App Store and only adds the app if it passes certain (unknown) security checks. On Android devices, the end user is also allowed to install apps from third-party markets. Especially in Asia, a lot of these markets emerged. Typically, these third-party markets pose a high risk to install malicious applications due to the fact that the market owners often do not adequately evaluate the applications they offer.

According to Felt et al. [13], mobile applications pose the following three types of threats:

Malware: Attackers want to gain access to the device by installing malware on it. The purpose is to steal data or damage the device. Malware is installed by tricking the user to

install a legitimate-looking application or in most cases to exploit a vulnerability on the device, e.g., a security flaw in the web browser.
Personal spyware: The purpose of spyware is to collect information about the victim and send them to the person who installed the spyware. Felt et al. argue that personal spyware will be installed on the victim's device by an attacker with physical access to the device and without his knowledge.
Grayware: The main purpose of grayware is to spy on users who installed the software on their own because they thought that it is legitimate software. Partly that is correct because the authors include real functionality as advertised. Nevertheless, they also collect information from the system such as the user's address book or this browsing history. The main goal is to collect information for marketing purposes, etc.

So overall, the threat landscape for Android is real and powerful analysis systems have to be developed to fight against these kind of threats.

2.2 Android system basics

We briefly introduce Android and its relevant parts for this paper in this section. For a thorough introduction, we refer to Six [29].

Android is based on Linux and therefore consists of the same core components as usual Linux distributions do. The core components are a (patched) Linux kernel, the Bionic *libc* and libraries like *WebKit*, *SQLite* and *OpenGL*.

The Android runtime environment consists of core libraries that provide most functionality provided by the core Java libraries. It additionally consists of the Dalvik Virtual Machine, which is responsible for running Android applications in the operating system.

Applications are written in the Java language, and each application is executed in its own Dalvik VM. This VM runs dalvik-dex code, which is translated from Java byte code. Dex code is an optimized bytecode suited for mobile devices; the biggest difference is that dex code is *register based* instead of *stack based*, as is "traditional" Java bytecode.

One relevant feature of the Dalvik VM for this paper is the ability that applications written in the Java language can additionally access native libraries through the *Native Development Kit* (NDK), which makes use of the *Java Native Interface* (JNI). Developers may move performance critical operations to native libraries (shared objects in the *ELF* format), which are then directly called from running dex code. Due to being executed natively and possibly faster, the code runs outside the Dalvik VM directly on the processor of the smartphone or emulator.

The inclusion of native code within Android applications does not alter the Android security model. The same architectural separations between apps and the well-known Android permission model is enforced regardless of the type of application.

3 Mobile-Sandbox: architecture and implementation

In order to determine whether an app is malicious or not, it needs to be analyzed with great effort. Its attributes as well as the function range need to be documented. Within this section, we describe the process of our automated analysis system (for a schematic overview see Fig. 1). The analysis process has been divided into two parts. We first discuss the static analysis in Sect. 3.1. The results of the static analysis are used to guide the following dynamic analysis, which is described in Sect. 3.2. The dynamic analysis automatically executes the apps on a modified Android system with the help of the Android emulator.

3.1 Static analysis

Our static analysis consists of several modules. In order to gain a first impression of the application that should be analyzed, the corresponding hash value is matched against the VirusTotal database. In this step, our system compares the md5 hash of the analyzed application with all hashes in the VirusTotal database, and if the hash can be found, then it calculates the detection rate (the number of tools that had classified the application as malicious divided through the number of anti-virus tools that had analyzed the given application). This received detection rate is stored for the report. However, it does not play a vital role within further processing as it is only there for a human investigator to get additional indicators if an application is malicious or not.

Afterward, the application is extracted—with the help of the tool `unzip`—in order to get access to its components; this is required for further analysis. As a following step, we analyze the Android manifest to get a listing of all required permissions. For this reason, we use the tool `aapt` being delivered with the Android SDK [16]. While parsing the manifest, we extract the intents as well as the services and receivers for further analysis, too. We also read out the SDK-version; this is another important detail to assure that only applications being compatible to the provided Android system are passed on to dynamic analysis (all the other applications will not be analyzed by our dynamic system and will be classified based on static analysis and machine-learning results only).

Now, the Dalvik byte code that is stored in the `classes.dex` file is converted to smali [15] to allow better automated parsing. We can determine and filter the embedded advertising networks from the resulting files and their directory structure as not to dilute the results of the analysis. This is done to not falsify the results because some advertis-

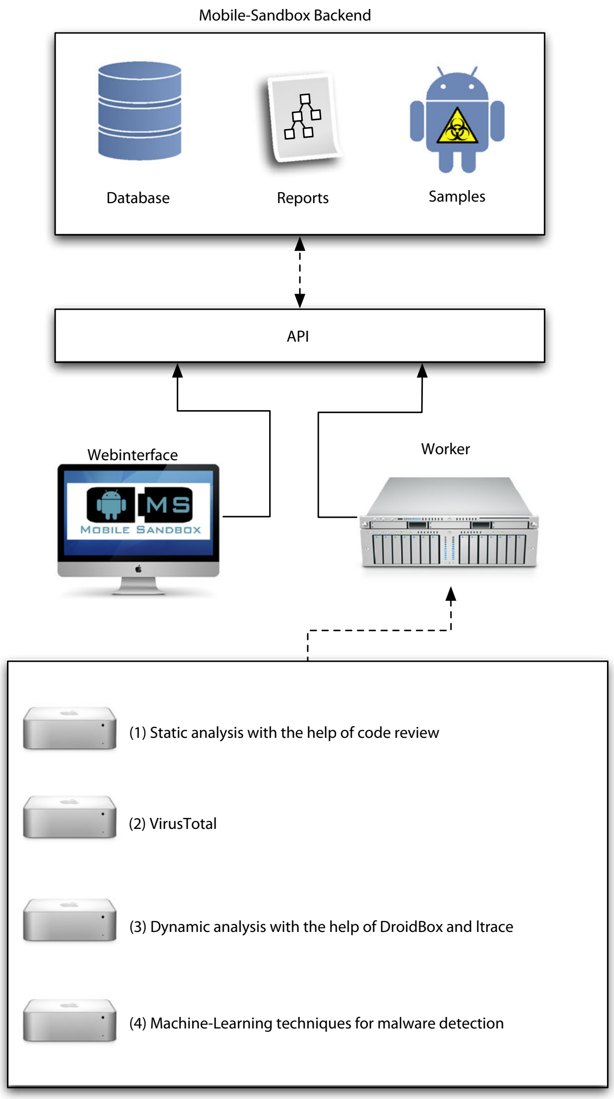
Mobile-Sandbox Backend
Database
Reports
Samples
API
Webinterface
MOBILE SANDBOX
Worker
(1) Static analysis with the help of code review
(2) VirusTotal
(3) Dynamic analysis with the help of DroidBox and ltrace
(4) Machine-Learning techniques for malware detection

Fig. 1 Schematic overview of the MobileSandbox's components

ing network use similar API calls than less evolved malware. Afterward, the complete smali code is searched for potentially dangerous functions and methods. Here, we take care of calls that can be found frequently, and in particular within malware, we used empirical values from our analysis of more than 300,000 samples to determine these calls. This includes, for example:

– `sendTextMessage()`: This call is responsible for the sending of SMS messages.
– `getPackageInfo()`: With the help of this method, malware often searches for installed AV products.
– `getSimCountryIso()`: This call is used to find out in which country the user currently resides. This is important in case of malware to contact the right premium services.
– `Ljava/lang/Runtime;->exec()`: Executes the specified command in a separate process. In case of malware, the commands are often 'su' or 'chmod'.

Moreover, we look for calls of the available encryption libraries. With this step, we try to gain deeper knowledge of the use of encryption and obfuscation within the applications. During the code-review, we try to recognize the functions and methods that normally need a permission for their error-free execution. For this reason, we refer to data of Felt et al. [14], which we translated from Java to smali. With the help of the gathered list, we can now compare whether the app is over- or underprivileged.

Over- and underprivileged apps are often used in correlation with each other or with apps that have the correct relation between permissions and function calls. For example: if you have an application `A` that is allowed to connect to the Internet (corresponding permission is requested from the user), but doesn't use this functionality, this app is overprivileged. Another application `B` is able to read all your contacts from your local address book, but is not able to send this data to a remote server as the needed functionality is missing. If a user now has both of these apps installed on his device, application `B` could send the sensitive data from the address book via an intent to application `A`, which then sends this data to a remote server. This example should demonstrates why overprivileged apps can represent a risk to a user. A similar risk goes out from underprivileged applications.

As an ongoing step, the source code is searched for statically coded URL's with the help of the following simple regular expression:

http [s]?://(?:[a−zA−Z]|[0−9]|[$−_@.&+]|[!∗\(\),]
|(?:%[0−9a−fA−F][0−9a−fA−F]))+.

We extract all implemented timers and broadcasts the app is waiting for, as a preparation for the dynamic analysis. Timers and broadcasts are event triggers for certain code to be executed in Android. We analyze these mechanisms to either trigger the corresponding events or wait for a specified time period in the ongoing dynamic analysis to improve the coverage of executed code. By this, we assure for example that the analysis is not stopped before a timer has expired. This is a common problem in Windows-based dynamic analysis [35].

At the end of the static analysis, a XML file is created, containing all data generated during the previous steps.

3.2 Dynamic analysis

While certain types of malicious behavior can already be recognized through static analysis, many kinds of malware can only be reliably detected by looking at its runtime behavior.

3.2.1 Building blocks

To perform a dynamic analysis of the application in question, we rely on the Android emulator provided by Google [2]. This software simulates a full ARMv7 device with key peripherals such as GSM module and touchscreen, thereby allowing to run unknown apps in a safe environment on a host computer. As an added benefit, the emulator can be reset to its previous state after an app has been tested by simply replacing the system image files with the original ones. Figure 2 gives an overview of the integration of the emulator into the entire Mobile-Sandbox framework.

Since the "stock" emulator offers only limited logging capabilities, we have chosen the well-known *TaintDroid/DroidBox* [12,21] system as basis for our dynamic analyzer. TaintDroid focuses on providing real-time privacy information to a user on a private device, while DroidBox builds on this work by logging all data accessed by the app to the system log, thereby creating a comprehensive picture of the app's runtime activities. This includes data read from and written to files, sent and received over the network, SMS messages sent, and many more.

While TaintDroid supports Android up to version 4.1.x, thereby covering at least 90 % of the device market as of January 2013, DroidBox only supports Android up to version 2.1, which can be considered severely outdated. Therefore, we updated the DroidBox patchset in a first step to work with TaintDroid 2.3.4, including some additional enhancements such as UDP support in the process and are going to update DroidBox to work with version 4.1.x at the time of this writing.

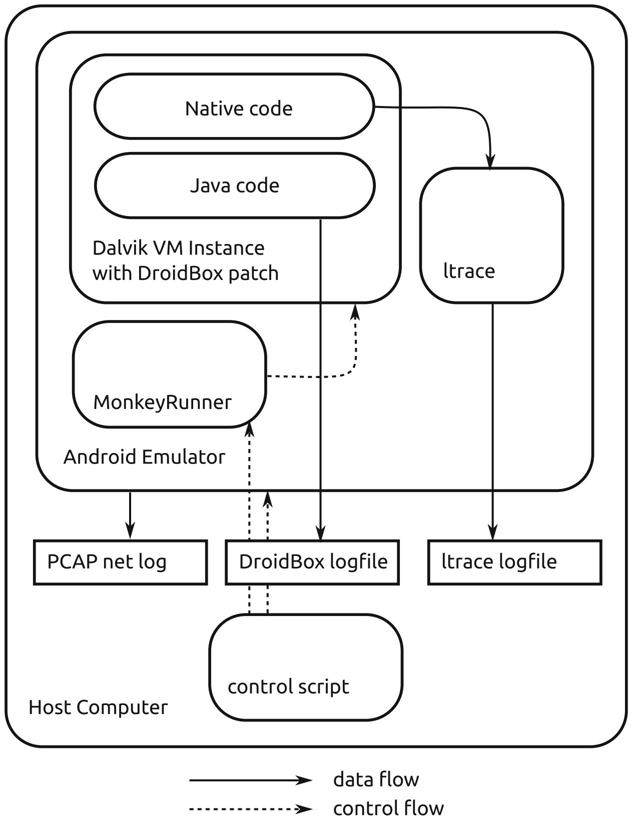

Fig. 2 Dynamic analyzer component overview

3.2.2 Tracking native code

However, this setup still has a "blind spot": since both TaintDroid as well as DroidBox are built on the Dalvik virtual machine used by Android to execute dex bytecode, only dex bytecode (in general translated from Java bytecode) can be traced by those tools. Native code executed using the JNI will not be visible. Since the introduction of the Android NDK, it is easily possible to call native code in external libraries.

In order to trace code included in such shared objects, we have included a modified version of the `ltrace` command, a common Linux debugging utility that intercepts library calls of a monitored application [32]. The modifications are necessary to run `ltrace` on the Android platform [11].

After the app to be examined has been started, an `ltrace` instance is launched and attached to the Dalvik process running the app in question. All native calls made into dynamically loaded shared objects are then logged to a separate file. To reduce the amount of log data and increase performance, functions commonly introduced by the NDK compiler are excluded from logging (like `_Unwind_Backtrace`). Another problem we were facing when logging all functions is that `ltrace` kills the app and the analysis ended in a kernel panic.

3.2.3 Network traffic

A third logging component that is already supported natively by the emulator is capturing of network traffic to a PCAP file. This common format can later be analyzed using tools such as WireShark.

From this captured network traffic, we extract information such as the hostname, the service port, and the data the app is sending. These kinds of data later help to classify whether the application is malicious, for example by checking whether the connected system's DNS name is a known malicious one.

Additionally, we convert the pcap file into plain text with the help of *derrick* [25]. This output is easier to read for humans when it is displayed on the website and it is also used for our machine-learning approach.

In summary, our setup produces three separate log files detailing the app's behavior (see also Fig. 2):

– *DroidBox logfile* containing important Java method calls and data from the Dalvik VM,

– *ltrace logfile* containing native method calls in shared objects using JNI and
– *network PCAP file* containing all data sent over the simulated 3G network.

3.2.4 User interaction

Another issue that has to be considered for dynamic analysis is that of code coverage. For most types of malware, simply launching the app will not trigger any malicious payload;it is necessary for the user to interact with the app and perhaps even confirm some malicious actions.

In order to test a significant fraction of the code paths present in the examined app, we use the *MonkeyRunner* toolkit provided by the Android SDK. Using this utility, it is possible to automatically send simulated interaction events, such as touchscreen contacts or key presses, to the tested app. Since MonkeyRunner does not take any UI elements into account, but rather produces random events, a sufficient number of events should be generated to make sure that most interaction elements have been triggered at least once.

3.2.5 Summary

In summary, the following steps are executed in order to dynamically analyze an app:

1. Reset emulator to the initial state.
2. Launch emulator and wait until startup is completed.
3. Install app to be analyzed using *adb*.
4. Launch app in a new Dalvik VM.
5. Attach *ltrace* to the VM process running the app.
6. Launch *MonkeyRunner* to generate simulated UI events.
7. Simulate additional user events like phone calls.
8. Launch a second run of *MonkeyRunner* that will run until all timers have passed.
9. Collect the Dalvik and ltrace log and the PCAP file.

The resulting log files of this process are inserted in our database. This database can be used to perform multiple other analyses.

3.3 Examples

To demonstrate the full range of functions of Mobile-Sandbox and the format of the log files, we now show some results of the log files that resulted from some applications we analyzed. We start with some information from our dynamic analysis that can be very useful when looking at encrypted data inside the application. In this case, the application has encrypted IMEI and IMSI numbers with the help of the DES algorithm before sending these data to a remote server. If you take a look at the network traffic, you would only see a package with encrypted data, but looking in the results from Mobile-Sandbox you get the decrypted data and the DES key that was used for encryption (see Listing 1).

```
Data — 357242043237517|310005123456789
Algorithm used — DES
Key — 7719681242490101965167123
```

Listing 1 Decrypted IMEI and IMSI and used encryption key

While executing the applications inside the emulator, we monitored several outgoing SMS messages (see Listing 2 for two examples).

```
Number: 84242 — Message: QUIZ
Number: 7132  — Message: 844858
```

Listing 2 Two examples for outgoing SMS messages

In addition, we found several kinds of data leakage. In Listing 3, we displayed the content of a file that was generated by the application.

```
File: /mnt/sdcard/Tencent/v1.log
Operation: write
Data: DeviceInfo [imei=357242043237517,
            telNum=, phModel=generic,
            sysSdk=10, RELEASE=2.3.4]
```

Listing 3 Data leakage to a file located on the unprotected SDcard

Within our network traffic analysis, we found a lot of privacy related data such as IMSI, IMEI or smartphone model, which was uploaded to remote servers or web services. One example for such an information leakage via HTTP POST can be seen in Listing 4.

```
<?xml version="1.0" encoding="utf−8"?>
<request>
   <version>1.15</version>
   <platform>2</platform>
   <pVersion>2.3</pVersion>
   <IMEI>357242043237517</IMEI>
   <simID>89014103211118510720</simID>
</request>
```

Listing 4 Information leakage found inside recorded network traffic

3.4 Malware detection using machine-learning techniques

During static and dynamic analysis, many features of an application are collected, which can be used to decide whether the application is malicious or benign. To achieve this, one would like to automatically identify specific patterns and combinations of extracted features that reflect malicious behavior. Machine-learning algorithms have already proven to solve this task very efficiently in other settings similar to ours [1,23,27]. In our setting, we apply a linear Support Vector in order to learn a classification model on a large dataset that contains malicious and benign samples. The resulting model can then be used to classify unknown applications as malicious or benign.

3.4.1 Embedding in vector space

Support Vector Machines operate on numerical vectors, and thus, one needs to find an appropriate vector representation for an application in order to apply this algorithm. To this end, we use a simple bag-of-words representation [26], where each possible feature string is associated with a certain dimension. The feature strings we use for the embedding are extracted statically as described in Sect. 3.1. The vector representation of an application can then be constructed by setting the respective dimension for each extracted feature to 1 and all remaining dimensions to 0.

3.4.2 Learning-based detection

Using this representation, applications sharing similar features lie closer to each other in the vector space than applications with few similar features. Ideally, malicious and benign applications only share few similar features and can thus be easily separated by a classifier, which considers geometrical information. In our case, a linear Support Vector Machine learns a hyperplane on a training set, which contains malicious and benign samples. The learned hyperplane separates both classes with a maximum margin and can afterward be used to classify unknown samples. An unknown application is then either classified as benign or malicious depending on which side of the hyperplane its vector is located.

We evaluate the detection performance of the classifier on a large dataset that contains 123,453 benign and 5,560 malicious samples [4]. For this purpose, we randomly split the dataset into a training and a test set. The detection model and respective parameters are determined on the training set, whereas the testing set is only used for measuring the final detection performance. We repeat this procedure 10 times and average results. Using the resulting classification model, we are able to detect about 94 % of the malware at a low false-positive rate of about only 1 %.

4 Evaluation

We now present an evaluation of our sandbox system. We analyzed the following aspects: correctness, performance, detectability, and scalability. At the end of this section, we also present a short case study of a malicious app using native calls.

Table 1 Overview of mobile malware used for evaluation

Malware family	Number of samples	Primary usage
Adsms	2	S
BaseBrid	5	I, B
SerBG	2	R, I, B
RootSmart	1	R, I, B, A
LeNa	1	R, I, B, A
Moghava	1	C
FakeInst	7	S
TapSnake	1	L

R gains root access, *C* compromises local storage, *S* sends SMS messages, *I* steals privacy related information, *B* botnet characteristics, *L* steals location data, *A* installs additional apps

4.1 Correctness

By correctness, we mean that an entry in the mobile-sandbox log file only appears if and only if the corresponding action was performed by the analyzed app. To check correctness, we chose 20 samples from a set of malicious applications that we collected from different sources. These samples represent different families of Android malware as shown in Table 1 meant to assure a wide coverage of malicious actions and different points in the development states of malware evolution. More specifically, we chose samples that hit the markets from mid 2010 until the beginning of 2012. The LeNa and RootSmart families use exploits and native calls, FakeInst and Adsms send premium SMS messages. The Moghava family acts only on the smartphone itself and modifies locally stored pictures, i.e., there is no malicious action observable that "leaves" the smartphone. The TapSnake family sends location information from the smartphone to a remote server.

Within this sample set, we consider RootSmart to be the most sophisticated malware sample which is, among other capabilities, also able to exploit the Android OS while Tapsnake is the simplest sample when looking at the techniques used for malicious behavior.

We manually inspected samples from all these families and consulted all available analysis reports by anti-virus companies and from other sources on the Internet. The resulting action sequences yielded the ground truth to which we compared the behavior the was output by Mobile-Sandbox. Overall, Mobile-Sandbox only detected actions that were part of the ground truth. However, after initial analysis, we failed to see certain behaviors that were described in the analysis reports on the Web. Later, we realized that the missing behaviors were due to missing external stimuli, i.e., remote servers of a botnet not being active anymore. These insights gave us additional confidence that Mobile-Sandbox was working correctly.

Table 2 Differences in build information between the emulator and a Samsung Galaxy S2

Build information	Emulator	Galaxy S2
Build.BOARD	Unknown	smdk4210
Build.DEVICE	Generic	GT–I9100
Build.MODEL	sdk	GT–I9100
Build.PRODUCT	sdk	GT–I9100
Build.TAGS	Test-keys	Release-keys
ro.kernel.qemu	1	0
ro.hardware	Goldfish	smdk4210

4.2 Performance

The performance of some parts of our system is still rather weak. During the evaluation for this paper, we measured runtimes between 9 and 14 min for the analysis of one single application. The system is running on an ESXi infrastructure on a server with Intel Xeon 2,4 GHz CPU and 48 GB of RAM.

In average, Mobile-Sandbox finishes the virus check within 3 s and the subsequent static analysis within additional 8–15 s. Afterward, the system needs about 2 min to reset and reboot a clean version of the emulator. After successfully booting the emulator, it takes another 2–6 min to install the application. This step depends on the file size of the application. The execution of the application and the MonkeyRunner scripts lasts another 6–10 min depending on the amount of user events and timers we want to trigger. After shutting down the emulator, the system needs additional 10 s for analysis of all log files and network traffic. The machine-learning techniques take less than 1 s, as we rate all relevant parts of the collected reports by reference to a generated model, which will be updated from time to time.

The performance can be enhanced tremendously by running multiple instances of the analysis frameworks, especially of the dynamic analysis, simultaneously.

4.3 Detectability

As we know from malware targeting the Windows environment, there are mechanisms to detect virtualized or sandboxed environments to make the analysis process of the malicious application more difficult or to act differently in these environments. Even if this behavior is not prevalent with Android at the moment, we think that this will change in the future. So an important security aspect is the detectability of our analysis platform.

A mechanism to detect the Android emulator deals with the specific builds of the operating systems that are used for it. An application querying this information can easily detect whether it is running inside an emulator or on a real device. Table 2 shows some system values that can be used for identification as they are sufficiently different from real smartphones. To prevent this detection mechanism, a custom-build of the Android system is required. In this build, we changed the first five variables from Table 2 to the values of a real Samsung Galaxy S2. Unfortunately, modifying the last two values can cause system crashes while running the emulator because there are some Android system services that rely on the fact that these values are set correctly. Another problem hiding the emulator is the fact, that Android launches the `qemud` and `qemu-props` daemons that offer emulation assistance to Qemu when running inside the emulator. Removing these two daemons is not feasible as they are needed to emulate the radio equipment.

Another weak point is Qemu itself. As emulated hardware always behaves differently from native hardware, an application is able to detect whether it is running inside an emulator by observing the behavior of certain performance aspects of the CPU. Raffetseder et al. [24] show multiple ways to detect the x86 version of Qemu. Similar techniques could also be applied to the ARM architecture to detect the corresponding Qemu version.

Additionally, we changed the default IMSI and IMEI of the emulator (originally both “0”) to random values that are consistent with regular IMEI and IMSI numbers. We did this modification to avoid emulator detection mechanisms that check for nonstandard or empty values in these device identifiers. We have seen this detection technique employed in various samples.

4.4 Scalability

Our last evaluation criterion refers to the scalability of Mobile-Sandbox, i.e., the question whether it can be used in large-scale analysis projects with several hundreds or thousands of apps.

4.4.1 Malware in third-party apps

To evaluate the scalability aspect, we collected 69,223 apps between December 2011 and March 2012 from the most important Asian markets by downloading them using the Android emulator in an automated fashion. We call this set of apps the “Asian set”.

We also received 6,162 Android malware samples from different families through anonymous uploads to our webservice [8] and through the VirusTotal Malware Intelligence Services (VTMIS) [19]. We call this set of samples the “malware set”.

We then used Mobile-Sandbox to automatically analyze 10,000 randomly chosen apps from the Asian set and 500 randomly chosen samples from the malware set. The analysis

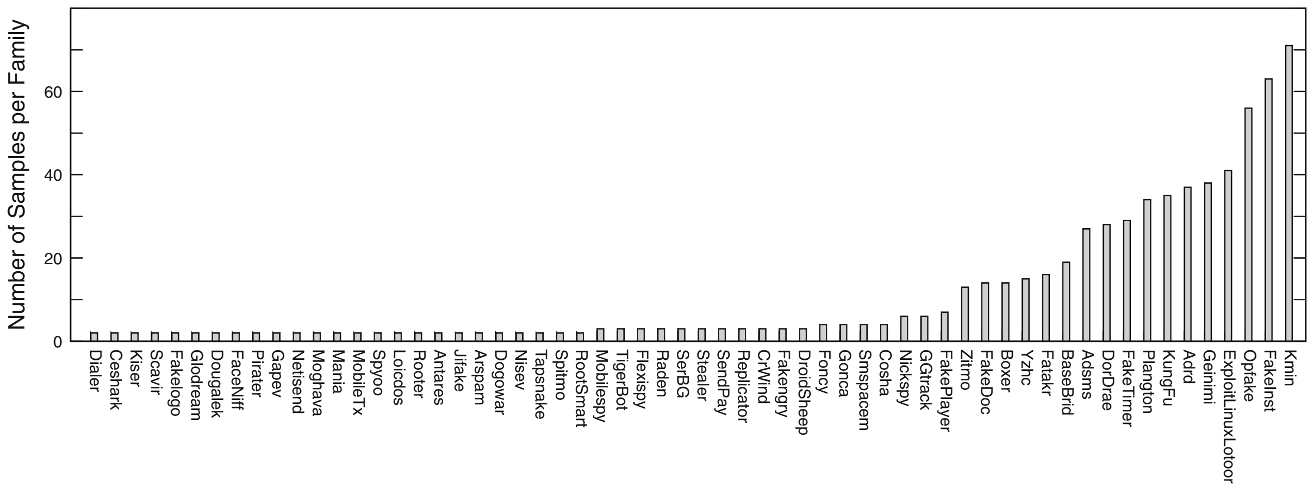

Fig. 3 Detected malware families and number of corresponding samples in the union of Asian and malware set

results were stored in a database on which we performed statistical analysis. These analyses were performed using a single installation of Mobile-Sandbox within a time span of 11 days.

Besides exhibiting that the use of Mobile-Sandbox can scale to several thousand apps, the statistical analysis provides some very interesting results. Considering the union of the Asian set and the malware set, we found 726 malicious samples according to our machine learning technique. Due to the fact that the malware set consisted of only 500 samples, there had to be 226 samples from the Asian set that were classified as malicious by our system.

Taking a deeper look at the distribution of the malicious applications, we noticed that about 26 % of the 726 samples belong to only three malware families (see Fig. 3 for a comprehensive overview). These families are Kmin, Opfake and FakeInst and their main functionality is sending premium SMS messages and, in the case of Kmin, information stealing.

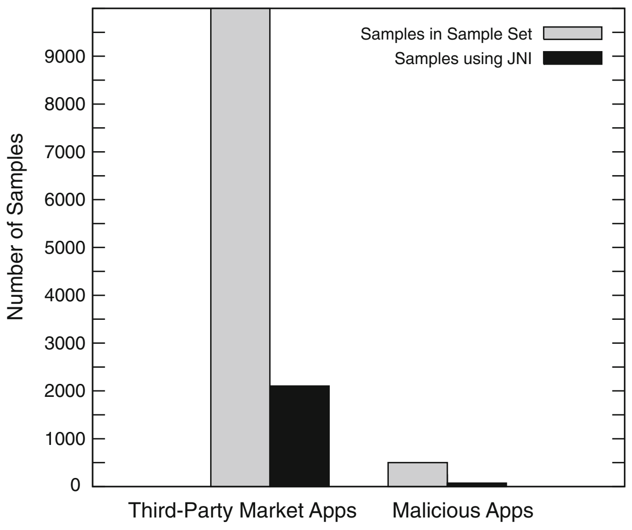

Fig. 4 Share of samples using JNI

4.4.2 Native calls

Another point of interest is the use of native interface calls inside Android applications. According to our analysis, about 21 % of the samples from the Asian set use native API calls. When one considers that the author of an app can potentially "hide" all malicious actions inside the native part of the application and that common tools are not able to trace or log this part of the application, the share of one out of four shows why it is so important to develop a system, which is able to log these events. Figure 4 also depicts the share of native calling apps in the malware set. Interestingly, this share is about 15 %. This means that the existence of native calls does not necessarily imply a higher probability that the app is malicious.

Within the Asian set samples that use native code, we found two samples that were hiding their malicious actions inside the native part of their code. When uploading these samples to VirusTotal at the time of this writing we got a detection rate of 0 %. This again makes clear how important it is to monitor and analyze native code.

4.4.3 Statistical implications

In summary, we found 2.26 % (226 out of 10,000) samples from the Asian set to be malicious. Recall that 2 out of these 226 were not detected by VirusTotal (0.02 % of the entire Asian set). But how representative are the results from our measurements? Since we have a rather large measurement base (number of measurements) and we have taken a random sample, we can apply quality assurance techniques from surveys performed in the area of social sciences by Groves [17,18].

In general, there are two quality criteria for statistical value estimations. The first is the *probability of error*, i.e., the probability that a statement is not true. In empirical research, the probability of error is acceptable if it is below 5 %. The second criterion is the *margin of error*, meaning the margin of percentages that the measurement could be different. Acceptable values are 5 % or below. Measuring a value of 75 with 5 % error probability and 5 % error margin, for example, means that with 95 % probability, the "real" value is between 70 and 80 %.

Following Groves [17,18], it is sufficient to analyze at least 664 samples to guarantee 1 % error probability and 5 % error margin. The measurements given above about the use of native code calls in the Asian set can therefore be generalized with high probability. To reach below a 1 % error margin in our finding regarding the part of malicious applications inside the Asian set (2.26 %) analyzed 10,000 samples [17,18].

4.5 Case study: a suspicious application using native calls

To show the full usefulness of our analysis engine, we now give a detailed analysis of one of the suspicious applications mentioned above which uses native code. The application appeared to be a map application, was not flagged as malicious by VirusTotal (detection ratio: 0/41).

During our dynamic analysis we found that the sample queried various privacy-relevant resources using the NDK. Overall, we rated the sample as suspicious due to the fact that it was collecting personal data.

In our static analysis, we saw that the application was requesting the following (unusually long) list of permissions:

- `ACCESS_COARSE_LOCATION`
- `ACCESS_FINE_LOCATION`
- `INTERNET`
- `VIBRATE`
- `READ_CONTACTS`
- `WRITE_CONTACTS`
- `RECEIVE_SMS`
- `READ_SMS`
- `WRITE_SMS`
- `SEND_SMS`
- `READ_PHONE_STATE`
- `ACCESS_NETWORK_STATE`
- `CHANGE_NETWORK_STATE`
- `WAKE_LOCK`
- `WRITE_EXTERNAL_STORAGE`
- `CHANGE_WIFI_STATE`
- `WRITE_SETTINGS`
- `SYSTEM_ALERT_WINDOW`
- `ACCESS_WIFI_STATE`
- `GET_TASKS`
- `CALL_PHONE`
- `BROADCAST_STICKY`
- `RECORD_AUDIO`
- `RECEIVE_BOOT_COMPLETED`
- `READ_PHONE_STATE`
- `GET_ACCOUNTS`

Although many of these permissions are required for a social map application such as the sample appeared to be, the high number of additional permissions such as `READ_PHONE_STATE` and `CHANGE_NETWORK_STATE` is questionable. Comparing the requested permissions with the permissions the app really needs for running, we noticed that the app is highly overprivileged.

In particular, it was interesting that, for example, the IMEI number was accessed by the regular Java API call `getDeviceID()` as well as by a native function `getImeiNumEv()`, which is not part of the Android API, but rather part of a native library packaged with the application. The same applied to the IMSI number (`getSubscriberID` vs. `getImsiNumEv()`).

Looking in the decompiled smali code, we found some more questionable code fragments like the following examples:

```
 – android/net/wifi/WifiManager;
->startScan
– android/app/ActivityManager;
->getRunningTasks
– android/media/MediaRecorder;
->setAudioSource
– android/telephony/SmsManager;
->sendTextMessage
```

Through our dynamic analysis we found that the app attempted to connect the mobile advertising network *flurry* and to various rather "dodgy" sites using the network protocols HTTP and HTTPS. Looking at this network traffic, we found various encrypted packages containing IMEI, IMSI and location data. Additionally, the application is listening for the `BOOT_COMPLETED` broadcast to start a background activity which is monitoring the location of the smartphone user and is listening to a remote server. Overall, we rated this application as highly suspicious.

5 Conclusions

In this paper, we proposed Mobile-Sandbox, a static and dynamic analyzer combined with machine-learning techniques for Android applications with the purpose to support malware analysts to detect malicious behavior. In the static analysis, we parse the application's Manifest file and decompile the application. In a further step, we determine whether the application is using suspicious looking permissions or intents. The second part of our sandbox performs the dynamic analysis where we execute the application in order to log all performed actions including those stemming from native API calls. Within our third set, we combine all of these results and try to detect malicious applications with the help of machine-learning techniques.

There are still many points to improve with Mobile-Sandbox, especially regarding the usability and compatibility with Android 4.x. For a better performance and usability, we need a substitute for MonkeyRunner as this solution is not precise and reliable enough. Or at least, we need to fix the crashes of MonkeyRunner. These improvements will probably also lead to a more reliable system.

For all the mobile users, we will provide an application for the Android platform that is able to check the status of already installed applications and is able to upload them directly into our sandbox for analyzing if no report is available.

Acknowledgments This work has been supported by the Federal Ministry of Education and Research (Grant 01BY1021 – MobWorm). We would also like to thank Felix Freiling, Konrad Rieck and Thorsten Holz for their valuable input and comments. Last but not least, we would like to thank the anonymous reviewers for proofreading our paper and for giving us helpful comments to improve this work.

References

1. Aafer, Y., Du, W., Yin, H.: DroidAPIMiner: Mining API-level features for robust malware detection in android. In: Proc. of International Conference on Security and Privacy in Communication Networks (SecureComm) (2013)
2. Android Developers.: Using the Android emulator. https://developer.android.com/guide/developing/devices/emulator.html. Jan 2012
3. Android Developers.: Android platform versions. http://developer.android.com/about/dashboards/index.html. Jan 2014
4. Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K.: Drebin: Efficient and explainable detection of android malware in your pocket. In: Proc. of Network and Distributed System Security Symposium (NDSS) (2014)
5. Bläsing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S., Albayrak, S.: An android application sandbox system for suspicious software detection. In: Proc. of the 5th International Conference on Malicious and Unwanted Software (MALWARE) (2010)
6. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for android. In: Proc. of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (2011)
7. Cristianini, N., Shawe-Taylor, J.: An Introduction to Support Vector Machines. Cambridge University Press, Cambridge (2000)
8. Department of Computer Science Friedrich-Alexander-University Erlangen-Nuremberg. Mobile-Sandbox. http://www.mobile-sandbox.com. Jan 2012
9. Desnos, A.: Androguard. http://code.google.com/p/androguard/. Jan 2011
10. Desnos, A., Gueguen, G.: Android: From reversing to decompilation. In: Proc. of Black Hat Abu Dhabi (2011)
11. Echtler, F.: ltrace for Android. https://github.com/floe/ltrace
12. Enck, W., Gilbert, P., gon Chun, B., Cox, L. P., Jung, J., McDaniel, P., Sheth, A.N.: TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In: Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI), October 2010
13. Felt, A., Finifter, M., Chin, E., Hanna, S., Wagner, D.: A survey of mobile malware in the wild. In: Proceedings of the 1st ACM wWorkshop on Security and Privacy in Smartphones and Mobile Devices, pp. 3–14. ACM (2011)
14. Felt, A.P., Chin, E., Hanna, S., Song, D., Wagner, D.: Android permissions demystified. In: Proc. of the 18th ACM Conference on Computer and Communications Security (2011)
15. Freke, J.: Smali—an disassembler for android's dex format. http://code.google.com/p/smali/. Sept 2009
16. Google Inc., Android SDK. http://developer.android.com/sdk/index.html. Oct 2009
17. Groves, R.M.: Research on survey data quality. Public Opin. Q. **51**(2), 157–172 (1987)
18. Groves, R.M.: Survey Errors and Survey Costs. Wiley, New York (1989)
19. Hispasec Sistemas S.L.: Virustotal malware intelligence services. https://secure.vt-mis.com/vtmis/
20. Hispasec Sistemas S.L.: Virustotal public API. https://www.virustotal.com/documentation/public-api/
21. Lantz, P.: Droidbox—android application sandbox. http://code.google.com/p/droidbox/. Feb 2011
22. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: Proc. of the 23rd Annual Computer Security Applications Conference (2007)
23. Peng, H., Gates, C.S., Sarma, B.P., Li, N., Qi, Y., Potharaju, R., Nita-Rotaru, C., Molloy, I.: Using probabilistic generative models for ranking risks of android apps. pp. 241–252 (2012)
24. Raffetseder, T., Kruegel, C., Kirda, E.: Detecting system emulators. In: ISC, pp. 1–18 (2007)
25. Rieck, K.: Derrick—a simple network stream recorder. https://github.com/rieck/derrick. Jan 2012
26. Salton, G., Wong, A., Yang, C.S.: A vector space model for automatic indexing. Commun. ACM **18**(11), 613–620 (1975)
27. Sarma, B.P., Li, N., Gates, C., Potharaju, R., Nita-Rotaru, C., Molloy, I.: Android permissions: a perspective combining risks and benefits. In: Proc. of ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 13–22 (2012)
28. Schmidt, A.-D., Bye, R., Schmidt, H.-G., Clausen, J., Kiraz, O., Yüksel, K., Camtepe, S., Sahin, A.: Static analysis of executables for collaborative malware detection on android. In: Proc. of the ICC Communication and Information Systems Security Symposium (2009)
29. Six, J.: Application Security for the Android Platform: Processes, Permissions, and Other Safeguards. Oreilly & Assoc Inc, Sebastopol (2011)
30. Spreitzenbarth, M.: Current Android malware. http://forensics.spreitzenbarth.de/android-malware/. Aug 2013
31. Spreitzenbarth, M., Freiling, F.C.: Android Malware on the Rise. Technical Report CS-2012-04, Dept. of Computer Science, University of Erlangen-Nuremberg, April 2012
32. The Debian Project. ltrace. http://anonscm.debian.org/gitweb/?p=collab-maint/ltrace.git;a=summary. Jan 2012
33. Vienna University of Technology.: Andrubis—analysis of android apks. http://anubis.iseclab.org. May 2012
34. Willems, C., Freiling, F.C.: Reverse code engineering—state of the art and countermeasures. it-Information Technology, pp. 53–63 (2011)

35. Willems, C., Holz, T., Freiling, F.C.: Toward automated dynamic malware analysis using CWSandbox. IEEE Secur. Priv. **5**(2), 32–39 (2007)
36. Xie, L., Zhang, X., Seifert, J.-P., Zhu, S.: pbmds: a behavior-based malware detection system for cellphone devices. In: Proc. of the Third ACM Conference on Wireless Network Security (2010)
37. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: Proc. of the 33rd IEEE Symposium on Security and Privacy (Oakland 2012), May 2012
38. Zhou, Y., Wang, Z., Zhou, W., Jiang, X.: Hey, you, get off of my market: detecting malicious Apps in official and alternative Android markets. In: Proc. of the 19th Annual Symposium on Network and Distributed System Security (2012)