Enhancing Security of Linux-based Android Devices

Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Jan Clausen, Ahmet Camtepe, and Sahin Albayrak
Technische Universität Berlin - DAI-Labor
Ernst-Reuter-Platz 7
10587 Berlin
Germany
{aubrey.schmidt, hans-gunther.schmidt, jan.clausen, ahmet.camtepe, sahin.albayrak}@dai-labor.de

Kamer Ali Yüksel and Osman Kiraz
Sabanci University Istanbul
Orhanli - Tuzla
34956 Istanbul
Turkey
{kamer, osmankiraz}@su.sabanciuniv.edu

This work was funded by Deutsche Telekom Laboratories.

1 Introduction

Our daily lives become more and more dependent upon smartphones due to their increased capabilities. Smartphones are used in various ways from payment systems to assisting the lives of elderly or disabled people. Security threats for these devices become increasingly dangerous since there is still a lack of proper security tools for protection.

Android emerges as an open smartphone platform which allows modification even on operating system level. Therefore, third-party developers have the opportunity to develop kernel-based low-level security tools which is not normal for smartphone platforms. Android quickly gained its popularity among smartphone developers and even beyond since it bases on Java on top of "open" Linux in comparison to former proprietary platforms which have very restrictive SDKs and corresponding APIs. Symbian OS for example, holding the greatest market share among all smartphone OSs, was closing critical APIs to common developers and introduced application certification. This was done since this OS was the main target for smartphone malwares in the past. In fact, more than 290 malwares designed for Symbian OS appeared from July 2004 to July 2008. Android, in turn, promises to be completely open source. Together with the Linux-based smartphone OS OpenMoko, open smartphone platforms may attract malware writers for creating malicious applications endangering the critical smartphone applications and owners' privacy.

In this work, we present our current results in analyzing the security of Android smartphones with a focus on its Linux side. Our results are not limited to Android, they are also applicable to Linux-based smartphones such as OpenMoko Neo FreeRunner. Our contribution in this work is three-fold. First, we analyze android framework and the Linux-kernel to check security functionalities. We survey well-accepted security mechanisms and tools which can increase device security. We provide descriptions on how to adopt these security tools on Android kernel, and provide their overhead analysis in terms of resource usage.

As open smartphones are released and may increase their market share similar to Symbian, they may attract attention of malware writers. Therefore, our second contribution focuses on malware detection techniques at the kernel level. We test applicability of existing signature and intrusion detection methods in Android environment. We focus on monitoring events on the kernel; that is, identifying critical kernel, log file, file system and network activity events, and devising efficient mechanisms to monitor them in a resource limited environment.

Our third contribution involves initial results of our malware detection mechanism basing on static function call analysis. We identified approximately 105 Executable and Linking Format (ELF) executables installed to the Linux side of Android. We perform a statistical analysis on the function calls used by these applications. The results of the analysis can be compared to newly installed applications for detecting significant differences. Additionally, certain function calls indicate malicious activity. Therefore, we present a simple decision tree for deciding the suspiciousness of the corresponding application. Our results present a first step towards detecting malicious applications on Android-based devices.

2 Related Work

Several publications were made in the field of smartphone malware detection and smartphone intrusion detection systems where tendencies can be seen that most promising approaches involve power usage data in order to detect attacks [1, 2, 3, 4]. Other approaches used feature vektor- and signature-based techniques in order to detect malware or anomalies [5, 6, 7, 8, 9, 10]. By now, no function call-based approaches for smartphones are known to the authors.

We present a method of static analysis of executables by disassembly. Essential characteristics like system and library functions are extracted and built the basis for identifying malware. Such an identification is done by a classifier, which is implementated using machine learning algorithms. Static analysis of executables is a well explored technique, recall for instance Christodorescu and Jha [11] or Zhang and Reeves [12] who propose such analysis to establish a similarity measure between two executables in order to identify metamorphic malware. Kruegel *et al.* describe static disassembly in [13]. Wang, Wu and Hsieh [14] present data mining methods to discriminate between benign executables and viruses, whose dynamically linked libraries and application programming interfaces are statically extracted. Support vector machines are used for feature extraction, training and classification. Eskin *et al.* [15] apply machine learning methods on a data set of malicious executables. Based on their data set they empirically show that the rule inducer RIPPER and naive Bayes estimators outperform simple signature-based scanner.

3 Android Security

Google Android[1] is a Linux-based operating system. And, as every other currently existing operating system, it offers at least basic functionalities and concepts to render it a fairly secure system. In following, we will take a deeper look into how Google implemented its security concept on Android from the OS-level perspective.

3.1 File Rights

All Java applications that are deployed on Android will be deployed in */data/<application_name>*. Upon creation of this directory, a new user and group are created and all files will given ownership to the new user. The user name will begin with "app_" and will be followed by a consecutive number, relying on which number the last user has received. The group name will be identical to the user name. Example:

```
drwxr-xr-x app_14 app_14 2008-09-17 14:26 com.android.sample
```

[1] http://code.google.com/android/

In this way, in case applications try to access files on system level, they will only be able to alter files with the same ownership. Access to different files is possible where altering external files is not possible.

3.2 System, userdata and SD-Card Image

Two main images are mounted upon start of the operating system (emulator): the system image and the userdata image. The system image is 65MB in size (approx. 21MB available) and is mounted to /system. It contains all Android system relevant files which can be:

- operating system relevant files like device files, drivers, libraries, system binaries etc.
- Android configuration files
- Android framework relevant files (e.g. android.awt, android.policy, services, etc.)
- Android base applications (e.g. Launcher, Browser, Phone, Contacts, etc.)

Additional applications will most likely face the problem that the available space will not suffice. If possible, parts can be relocated into the userdata or SD-Card image.

The userdata image is also 65 MB in size (approx. 40MB available) and mounted to */data*. It contains all user relevant files. New applications will most likely find their way into the userdata image. Additionally, there are reserved directories for DRM, log files, and more. Approx. 40MB available size give a bit space to play. But still though, it is a very limited resource.

The emulator offers the possibility to mount given SD-Card images into Android, to be specific, to */sdcard*. Google does not offer a given image, it has to be created. There is no given limitation in size, which, at first sight, seems to be a promising location to include applications that exceed given space in */system* or */data*. But, there is a major restriction for */sdcard*: all files will be created as user "system" (group: "system") with READ and WRITE permissions - no EXECUTE flag. Trying to change this will fail: changes to file permissions are simply not allowed on this image! Therefore, applications within the SD-Card image cannot be executed.

3.3 Application Signing

Android requires (Java) application signing; unsigned applications cannot be installed to the system. The user is able to use self-signed certificates to sign applications. No central certificate authority is needed. At the moment, it is not clear whether this will change or not. But for now, this type of signing does not provide an acceptable level of security.

4 Creating a Build Environment for Android

Android provides a complete operating system based on the ARM-Architecture. Compiling software for ARM requires a specific environment. In following sections, we will describe two different ways of compiling software successfully within an ARM-compatible environment.

4.1 Base environment

Ubuntu i686 GNU/Linux, a Linux-distribution provided and supported by Canonical, provides the basis for all further steps. Based on the Intel-architecture, a vast amount of tools, especially for creating and compiling software, can be obtained through Ubuntu's package repositories. Additional, non-standard, package repositories can be easily integrated.

4.2 GNU Toolchain for ARM Processors

`CodeSourcery.com` offers a cross-compile toolchain that can be used to cross-compile source code for ARM on various architectures other than ARM. Consisting of C/C++-compiler, linker, libraries, several tools for debugging and more, it offers everything required for compiling tools that can be executed in an Android environment. Providing the required information during configuration run (passing parameters to use a different compiler, compile for a different architecture, additional usual compile parameters), there is no difference between compiling (open) source code for ARM or for Intel. Once compiled, the software needs to be transferred to the Android environment in order to test its functionality. This might be, at some times, a tiring task. Therefore, a second way to compile source code for ARM is presented in the next section.

4.3 Scratchbox Cross-Compilation Toolkit

Scratchbox not only provides the necessary compilers and linkers, it also provides a complete environment simulating an ARM platform-based operating system. All tools compiled within this environment can be tested immediately giving a very fast feedback to the developer. Once, the Ubuntu package repository has been extended by the official Scratchbox repository, all necessary files for Scratchbox can be easily installed via Ubuntu's package management tools. Scratchbox offers a wide variety of possible compilers, in different versions and characteristics. After installation, a user has to be configured that will be used within the Scratchbox environment. Shortly after logging into the new environment, preliminary steps are required: select the desired compiler and add additional tools if wanted (strace, gdb). From now on, source code can be compiled as usual, no specific parameters have to be provided. The host and build type are distinguished automatically, standard locations for installing binaries, libraries, etc. are provided. As long as the given source code is ARM-compatible, it will most likely compile within scratchbox without any significant problems. Having successfully compiled all files, these can be packed into an archive for being transferred to the Android environment for further deployment.

4.4 Important Facts When Compiling for Android

Google provides an ARM Linux with a filesystem layout which greatly differs to usual Linux filesystem layouts:

- System relevant files are found in the System image, mounted to */system* (binaries are, for the most part, found in */system/bin*, libraries reside in */system/lib*, configuration files in */system/etc*, etc.)
- User data relevant files reside within the user data image, mounted to /data

Handling this change does not require too much adaption.

4.4.1 All-In-One Binary Toolbox

Furthermore, Google provides standard Linux tools with the help of the all-in-one binary *toolbox*. "toolbox" only offers a very restricted set of tools making it at certain times hard to accomplish standard procedures. Special care has to be taken here when including Shell scripts that rely upon various Linux system tools.

Installing *busybox*, also a all-in-one static binary offering numerous standard Linux system tools, helps greatly.

4.4.2 Location-awareness of Tools

Certain tools within Android are *location-aware*. A specific action, e.g. changing file permissions or ownership, will execute successfully without any further notice in /system. The same action, executed for files in your SD-Card-image will simply fail. This implies that tools can only be executed from within /system or /data. Adding tools via a freely resizable SD-Card-image will not be possible.

4.4.3 Disk space limitations

/data and */system* offer only very limited flexibility as they are both limited to a maximum filesystem size of 65Mbytes. While in a standard, untouched, Android Linux, there is about 40MB of space left within */data*, the System image, at the same time, offers only approximately 20MB for additional tools. This fact requires appropriate counter-measures when configuring given source code for compilation (e.g. ClamAV database needs to be placed in a different location as it exceeds the given 20MB on */system*).

4.4.4 Page alignment causes changes in linking

Of very high impact on the success of compiling software for Android is the fact that Google forces compatible binaries to not be page aligned for the text and data section. This requires changes in the way of linking object files. For self-written software, one can take precautious steps and react on this fact with compiling all shared libraries accordingly. For already existing source code, changing the linker's behavior can present a very tiring and, often, an even impossible task.

4.4.5 Compiling statically

Due to the different approach of linking, the only way to run open source software on Android without altering the source code is to compile the source code statically. The output binary will have only small dependencies to existing libraries making it relatively autonomous. For a fair amount of available open source software, this method has been executed successfully. Still though, tools like "iptables" or "Snort" will not accept this method and fail compiling.

5 Enhancing Security for Android-based Devices with Common Linux Tools

For improving the security of Android-based devices, additional tools might be installed to the Linux-layer of the operating system. Since Linux-based security research is mature, several useful open-source tools are available to the public. In the following, we will present a list of tools that is well-accepted among the Linux community. This list is categorized into fields of application. From each category we will choose an example tool for describing the main usage and installation to the emulator environment.

5.1 Interesting Tools for Android

This section includes tools that are well-known to the Linux community. We mainly focused on open source software that can be modified to run it on Android. A good source for finding security tools is the “Top 100 Network Security Tools” list from [16]. Obviously exhaustive tools have already been removed after first review. The categories of applications are: anti-virus, firewalls, rootkit detectors, intrusion detection system (IDS), and other useful tools. From each category one tools is chosen where we will give general public information on it. The categories and chosen tools can be seen on Table 1 in the summary. For each tool, we will describe whether it can be run on the current version of Android, what the file space costs are, and which problems appeared and how to solve them.

5.1.1 Anti-Virus Tool

> Clam AntiVirus[2] is an open source (GPL) anti-virus toolkit for UNIX, designed especially for e-mail scanning on mail gateways. It provides a number of utilities including a flexible and scalable multi-threaded daemon, a command line scanner, and advanced tool for automatic database updates. The core of the package is an anti-virus engine available in a form of shared library.

[2] `http://www.clamav.net/`

Android Compatibility: Works (not all parameters tested)
Problems, solutions, and size:

- Static compilation required (LDFLAGS="-static")
- Dependent on static compiled version of "zlib" (zlib-1.2.3)
- Total size of all ClamAV relevant files (approx. 28MB) exceeds available size in System image (21MB). ClamAV virus signature database needs to be placed in a different location.
- Size (approx.): 11140 KB libraries and binaries (*/opt*), 17324 KB database (*/data*)

5.1.2 Firewall

netfilter[3] is a set of hooks inside the Linux kernel that allows kernel modules to register callback functions with the network stack. A registered callback function is then called back for every packet that traverses the respective hook within the network stack.

Android Compatibility: Netfilter kernel extension can be provided by modified kernel, iptables does not work
Problems:

- Kernel needs to be recompiled from source. Sources can be freely downloaded from Android Project website. Enable NETFILTER in kernel configuration and recompile!
- "iptables" cannot be compiled due to page alignment issues (requires statically compiled parts of libc which Android does not provide)

5.1.3 Rootkit Detectors

Chkrootkit[4] scans for signs of rootkits, worms and Linux Kernel Module (LKM) trojans. Possible methods are:

- Inspect binaries
- Check logs for suspicious entries
- Check network interfaces for promiscuous mode
- Look for hidden files in */proc*s

Android Compatibility: Works with minor dependencies
Problems, solutions, and size:

- Static compilation required (LDFLAGS= "-static")
- Requires "netstat" (provided by "busybox")
- Requires standard directories (*/lib*, */etc*, etc.) which can be created by symbolic links pointing to the correct Android directories
- Size (approx.): 588 KB

[3] http://www.netfilter.org/
[4] http://www.chkrootkit.org/

5.1.4 Intrusion Detection

Snort[5] is a lightweight network intrusion detection and prevention system that excels at traffic analysis and packet logging on IP networks. Through protocol analysis, content searching, and various pre-processors, Snort detects thousands of worms, vulnerability exploit attempts, port scans, and other suspicious behavior. Snort uses a flexible rule-based language to describe traffic that it should collect or pass, and a modular detection engine. A graphical user interface is given with the Basic Analysis and Security Engine (BASE), a web interface for analyzing Snort alerts.

Android Compatibility: Snort will not compile statically due to static libc requirements
Problems:

- Dependencies to libpcap, libdnet, libnet, pcre and iptables (all as statically compiled solutions)
- Requires statically compiled libc parts which are not available on Android

5.1.5 Other Useful Tools

Busybox

BusyBox[6] combines tiny versions of many common UNIX utilities into a single small executable. It provides replacements for most of the utilities you usually find in GNU fileutils, shellutils, etc. The utilities in BusyBox generally have fewer options than their full-featured GNU cousins; however, the options that are included provide the expected functionality and behave very much like their GNU counterparts. BusyBox provides a fairly complete environment for any small or embedded system.

Android Compatibility: Works
Problems, solutions, and size:

- Static compilation required (LDFLAGS="-static")
- Size (approx.): 1712 KB

Bash

Bash[7] is the shell, or command language interpreter, that will appear in the GNU operating system. Bash is an sh-compatible shell that incorporates useful features from the Korn shell (ksh) and C shell (csh). It is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard. It offers functional improvements over sh for both programming and interactive use. In addition, most sh scripts can be run by Bash without modification.

Android Compatibility: Works
Problems, solutions, and size:

- Static compilation required (LDFLAGS="-static")
- Busybox
- Size (approx.): 1780 KB

[5] http://www.snort.org/
[6] http://www.busybox.net/
[7] http://www.gnu.org/software/bash/

strace

strace[8] is a useful diagnostic, instructional, and debugging tool. System administrators, diagnosticians, and trouble-shooters will find it invaluable for solving problems with programs for which the source is not readily available since they do not need to be recompiled in order to trace them. Students, hackers and the overly-curious will find that a great deal can be learned about a system and its system calls by tracing even ordinary programs. And programmers will find that since system calls and signals are events that happen at the user/kernel interface, a close examination of this boundary is very useful for bug isolation, sanity checking, and attempting to capture race conditions.

Android Compatibility: Works
Problems, solutions, and size:

- Static compilation required (LDFLAGS= "-static")
- Requires patch "strace-fix-arm-bad-syscall.patch"
- Size (approx.): 1012 KB

OpenSSH

OpenSSH[9] is a FREE version of the SSH connectivity tools that technical users of the Internet rely on. Users of telnet, rlogin, and ftp may not realize that their password is transmitted across the Internet unencrypted, but it is. OpenSSH encrypts all traffic (including passwords) to effectively eliminate eavesdropping, connection hijacking, and other attacks. Additionally, OpenSSH provides secure tunneling capabilities and several authentication methods, and supports all SSH protocol versions.

Android Compatibility: Works with minor dependencies
Problems, solutions, and size:

- Static compilation required (LDFLAGS= "-static")
- Requires static version of OpenSSL library
- Size (approx.): 10688 KB

Nmap

Nmap[10] ("Network Mapper") is a free and open source utility for network exploration or security auditing. Many systems and network administrators also find it useful for tasks such as network inventory, managing service upgrade schedules, and monitoring host or service uptime. Nmap uses raw IP packets in novel ways to determine what hosts are available on the network, what services (application name and version) those hosts are offering, what operating systems (and OS versions) they are running, what type of packet filters/firewalls are in use, and dozens of other characteristics. It was designed to rapidly scan large networks, but works fine against single hosts. Nmap runs on all major computer operating systems, and both console and graphical versions are available.

Android Compatibility: Works
Problems, solutions, and size:

[8] `http://sourceforge.net/projects/strace`
[9] `http://www.openssh.com/`
[10] `http://nmap.org/`

- Static compilation required (LDFLAGS= "-static")
- Requires static version of OpenSSL library
- Requires libpcap, liblua and pcre (included in Nmap sources, needs to be activated by Nmap configuration parameters before compilation)
- Size (approx.): 5444 KB

5.2 Summary

In this section, we pointed out which current Linux applications can be installed to the Android emulator environment. The results will most likely be applyable to real Android devices. The main problems in compiling Linux applications for Android can be seen in the lack of space, the missing and modified libraries, as well as in the static compilation dependencies. A wrap up of our results can be found on Table 1.

Table 1: Tested Security Tools

Category	Chosen Tool	˜ Size	Works on Android
Anti-Virus	ClamAV	28.5 MB	●
Firewall	iptables	-	-
Rootkit Detector	Chkrootkit	0.58 MB	●
Intrusion Detection	Snort	-	-
Other Useful Tools	Busybox, Bash, openssl, ...	var.	●

6 Enhancing Security with a Self-built Intrusion Detection System

In this section we present our first results in creating an Intrusion Detection System for the Android platform. Therefore, we present our architecture and corresponding detection system. As an example detection approach, we will present our first results using static function call analysis for detecting malware.

6.1 Architecture

Figure 1 shows the architecture of the monitoring and detection client. The bottom-up view on it starts with the Linux operating system level generating signals received by the actual monitoring components. The Linux application level provides all the functionality needed for monitoring and storing device and operating system information. On Java application level anomaly detection, detection collaboration, and detection response are realized where the corresponding states can be visualized in a user interface.

6.1.1 Linux Operating System Level

The Linux operating system level provides events that are recognized by the monitoring system. These events are initiated by kernel or file system changes.

6.1.2 Linux Application Level

The monitoring architecture on Linux application layer consists of two programs: the monitoring application and the control daemon. The control daemon is responsible for checking the status and persistence of the monitoring application. The monitoring application extracts information (features) from the Linux

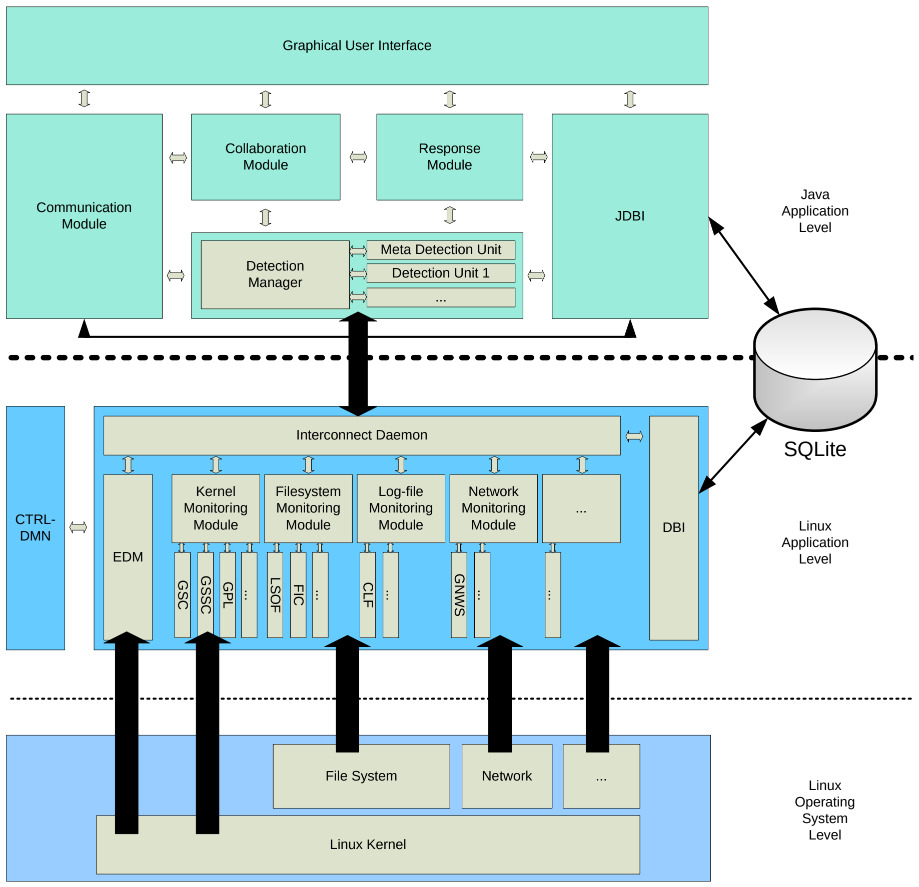


Figure 1: Monitoring and Detection Client Architecture

kernel and file system. These features are used by the detection for creating a sense of normality. Therefore, the features contain information about the hardware and software states of the device. It has an generic and extensible design for modifying it to corresponding needs.

Interconnect Daemon This is the main module of the monitoring application. It is triggered and controlled by the event detection module for generating vectors containing features.

Event Detection Module (EDM) This is an essential component of the monitoring system. It recognizes changes in the kernel and file system and generates corresponding events, e.g. new process is started. Basing on these events, features are extracted that can vary in their content and size. Each feature is marked with a time stamp and event for later processing.

Kernel Monitoring Module This module extracts kernel-based features. Examples for this are process lists, system call traces, and symbol tables.

Filesystem Monitoring Module This module extracts and verifies information on files. Examples for this are a list of open files or an integrity check on predefined files.

Log-file Monitoring Module Since Android and many applications support logs, this module extracts information on changes and existence of these.

Network Monitoring Module This module can extract information on current network configurations, configuration changes, network status, and network traffic.

Database Interface (DBI) This interface provides access to the Android SQLite database from Linux application level. It is mainly used to store the feature vectors created by the event detection module.

6.1.3 Java Application Level

The monitoring and detection architecture on Java application layer realizes several tasks for anomaly detection, detection collaboration, and detection response.

Detection Module The *detection module* runs light-weight detection algorithms basing on feature vector excerpts from the database. It consists of a detection manager coordinating a variable amount of detection plug-ins. These plug-ins are instances of detection algorithm that on the one hand can analyze feature vectors and on the other hand can analyze results from different detection algorithms. Whenever cooperative detection algorithms are used, this module can additionally trigger the collaboration module.

Collaboration Module The *collaboration module* provides the means to enable detection as well as response in a collaborative manner as an API. Therefore, the collaboration module stores the node configuration of the device in a dedicated data model. Based on this model, *interests* for the collaboration can be defined that are matched against other node configurations. Thus, partners for the purpose of collaboration are found and communicated with via the *communication module*.

Response Module This module enables countermeasures to detected incidents.

Communication Module For exchanging feature vectors with the remote server or collaborative peers, this module provides suitable functions and network access.

Java Database Interface (JDBI) This interface provides access to the Android SQLite database from Java application level. It is mainly used to extract feature vectors and detection results recorded by the system.

Graphical User Interface This module visualizes current monitoring, detection, collaboration, and response status.

6.2 Detecting Anomalies

6.2.1 Approach

An open system like Android requires protection against unwanted software and intrusion. In general, there are two principal techniques handling this, namely misuse detection and anomaly detection. The former method is intended to recognize known signatures of malware and attacks, the latter to determine the degree of normality of some observables. Since there is no malware existent for the Android device, our focus is set on *anomaly detection*. Anomaly detection can be used to identify new and unknown attacks, which in turn can be used on- and offline to generate signatures for fast detection in the future. Note that the detection architecture does not need to be changed for misuse detection.

The question arises what normality means. In our approach we distinguish an individual and a common sense of normality. Either are learned statistically and each device can check a system state according to both measures.

When constructing a detection mechanism for a mobile device such as Android, the computational costs has to be kept acceptable due to the limited resources and the need of energy saving. Thus *energy efficiency* is a guide line for the architecture. Taking this into account, complex computational task and the storage of huge data sets is outsourced to an *external server* and the on-device detection algorithm is kept relatively simple. Since each detection requires energy, the system integrity should not be checked more often then really necessary, i.e. only on certain occasions. Hence, an *event-based* approach seems more reasonable than, e.g. a time-periodical one. Furthermore, neighbor devices are taken account in order to collaborate and exchange data in the existence of an ad-hoc network.

6.2.2 Detection Mechanism

According to our approach five major tasks have to be handled:

1. Event perception, which is done by an *event sensor* (event detection module (EDM)).
2. System monitoring, to gain informations about some system observables (features) when required. For each class of event there is an adequate monitoring module, recall Figure 1, the entirety of those we will call *system monitor*.
3. Detection, i.e. analyzing system features and assigning a level of normality, done by the detector, which consists of a *detection manager* and event-specific *detection units* and *meta detection units*.
4. Learning, which the external server is responsible for.
5. Collaboration, which is used in the absence of external server or for reducing the load from the external server.

6.2.3 Architecture

Figure 2 outlines the architecture of the detection. The detection manager is a daemon, which can be implemented as an Android activity. It is set on auto start and on the highest priority level. The system is prevented from stopping the detection manager via the method `setPersistent()`. In this way, it is assured that it runs permanently in the background. Normally, an activity should not be set persistent since then it blocks system capacities.

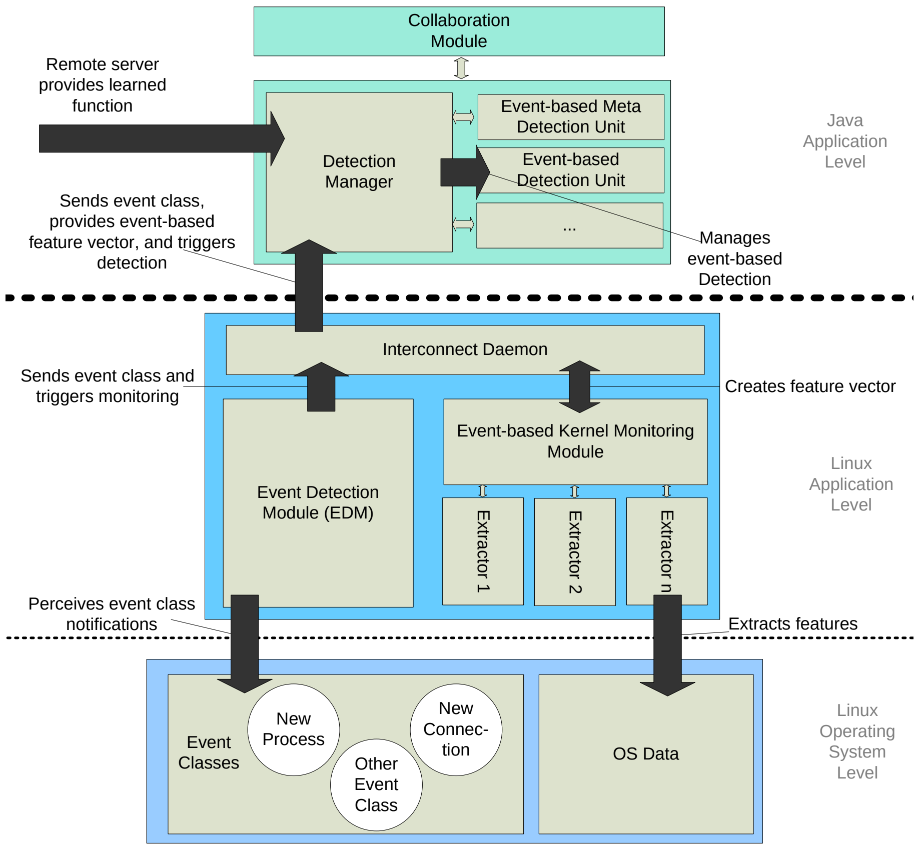
The jobs which have to be accomplished by this unit is receiving signals sent by the event detection module and starting a corresponding detection unit. The latter are implemented as sub-activities and assign to each feature vector a level of abnormality and return them (in a bundle) to the detection manager. If it exceeds a predefined threshold, the detection manager will alert the user via GUI.

The external server does the hard work of statistical learning. The accumulated training data is sent from the database of a mobile device to this server, and in turn the server provides updated parameters for the detection to the mobile device. For a more detailed view on learning see 6.2.4.

Let the *interaction* of these units be described by an example. Assume that one of the events we described occurs, say a new process is being launched. This event is sensed by the event detection module, which informs the system monitor and the detection manager immediately about which kind of event has occurred. The system monitor then extracts some (event-specific) system features, in this case the sequence of system calls caused by this new process along with CPU/memory utilization and other process data. Meanwhile the detection manager has started an event-specific detection unit, i.e. the detection unit corresponding to the "process-started event". This detection unit evaluates then the level of alert from the feature vector provided by the system monitor.

6.2.4 Server supported learning

Whatever reasonable learning technique is chosen, the computational costs for training cannot be carried by the mobile device in almost all cases. Hence, the training data, gained from monitoring, is gradually stored in a database and — after a certain of data amount has been accumulated — sent to a server, where the individual detection parameters are evaluated. Training data is separated according to event class so that event specific detection parameters are determined and sent back to mobile afterwards. Each detection unit attains in this way an understanding of normal system behavior which follows each specific event. Furthermore the server also calculates a common sense of normality based on the broad statistical data of all users and makes these common parameters available for the detection units of each user. The



Collaboration Module
Remote server provides learned function
Detection Manager
Event-based Meta Detection Unit
Event-based Detection Unit
...
Java Application Level
Sends event class, provides event-based feature vector, and triggers detection
Manages event-based Detection
Interconnect Daemon
Sends event class and triggers monitoring
Creates feature vector
Event-based Kernel Monitoring Module
Event Detection Module (EDM)
Extractor 1
Extractor 2
Extractor n
Linux Application Level
Perceives event class notifications
Extracts features
Event Classes
New Process
New Connec-tion
Other Event Class
OS Data
Linux Operating System Level

Figure 2: Architecture of Detection Mechanism

reason for that is that user behavior might switch abruptly if, e.g., a new application has been recently installed. Then the detection unit will state a high *individual* level of alert whereas it will claim that the system behavior is fairly normal relative to other users, since some of which might have already worked with this new application.

6.3 Static Function Call Approach

As an example of the detection concept presented in the foregoing section we turn to a rather concrete scenario. The considered event class is now the execution of a binary and the system observables are the *static function calls* of this executable. Information about functions which might be called by an executable is gained by means of disassembling, which is done by the Linux commands `readelf` and `objdump` in our case, recall Linux `man` pages. We show that normal executables are distinguishable from abnormal, which are represented by linux malware, on the basis of function appearance in the static table. In our approach the set of normal executables consists of 94 Android linux commands, mostly found in `/bin`, and the set of abnormal executables consists of linux malware, via Google search we found 240 of the latter.

We induced decision rules in the following way. First, the set of functions, appearing in our normal and malicious set, is reduced by taking only those functions which appear in the malware set and normal set. This is done to exclude any Android specific calls, which are frankly not called in the Linux malware. Second, we apply principal component analysis to reduce further the number of functions we will look at. Third, decision rules are created based on the remaining functions. With the help of the celebrated decision tree learner ID3, developed by Quinland [17], we created two efficient and accurate decision rules based on different function sets, see figures 3 and 4.

The accuracy parameters are determined by stratified 10 fold cross validation. The rate of malware detection are higher than 95% for both decision trees; the rate of false positives, i.e. normal executables

```
_edata = y
|  gethostbyname = y
|  |  sigaction = y: normal
|  |  sigaction = n: malicious
|  gethostbyname = n
|  |  fork = y
|  |  |  strerror = y
|  |  |  |  getgrgid = y: malicious
|  |  |  |  getgrgid = n: normal
|  |  |  strerror = n: malicious
|  |  fork = n: normal
_edata = n
|  exit = y: malicious
|  exit = n
|  |  fprintf = y: malicious
|  |  fprintf = n
|  |  |  uname = y: malicious
|  |  |  uname = n
|  |  |  |  execv = y: malicious
|  |  |  |  execv = n
|  |  |  |  |  malloc = y: malicious
|  |  |  |  |  malloc = n
|  |  |  |  |  |  putchar = y: malicious
|  |  |  |  |  |  putchar = n
|  |  |  |  |  |  |  memmove = y: malicious
|  |  |  |  |  |  |  memmove = n: malicious
```

Figure 3: Decision tree 1. y means that the function appears in the static table of an executables, n that not.

erraneously classified as malicious, is 13% for the first and 11% for the second decision tree respectively.

7 Conclusion and Future Work

In this paper, we gave an first overview on how Android-based smartphones can be secured. Therefore, we presented Android security mechanisms, listed tools that can be added, and presented our own intrusion detection system approach. We additionally presented first results of applying static function call analysis to Android.

Android represents a great opportunity for researching security aspects on mobile devices, like smartphones. Since it will be set open source as soon as the first devices are released, this is the first time that most functionalities and APIs will be available to common developers. This will most probably lead to benign and malicious research activities hopefully resulting in an even more secure smartphone plattform.

Since real Android devices are not available yet, most of our findings have to be approved as soon as first devices are released. But from this point of view, an Android user has various possibilities to increase out-of-the-box security of his handset. One major issue will be the limited space for installations. It is not clear yet, whether the space can be extended to an attached SD-card, or not. Therefore, building some light-weight detection system that take the resource constraints into account is a good idea to partly solve the space problem.

Acknowledgments:

We appreciate the fruitful discussion at DAI-Labor / CC-Security.

```
__bss_start = y
|   gethostbyname = y
|   |   sigaction = y: normal
|   |   sigaction = n: malicious
|   gethostbyname = n
|   |   fork = y
|   |   |   strerror = y
|   |   |   |   getgrgid = y: malicious
|   |   |   |   getgrgid = n: normal
|   |   |   strerror = n: malicious
|   |   fork = n: normal
__bss_start = n
|   printf = y: malicious
|   printf = n
|   |   fprintf = y: malicious
|   |   fprintf = n
|   |   |   execv = y: malicious
|   |   |   execv = n
|   |   |   |   memmove = y: malicious
|   |   |   |   memmove = n
|   |   |   |   |   perror = y: malicious
|   |   |   |   |   perror = n: malicious
```

Figure 4: Decision tree 2

References

[1] D. C. Nash, T. L. Martin, D. S. Ha, and M. S. Hsiao, "Towards an intrusion detection system for battery exhaustion attacks on mobile computing devices," in *PERCOMW '05: Proceedings of the Third IEEE International Conference on Pervasive Computing and Communications Workshops*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 141–145.

[2] H. Kim, J. Smith, and K. G. Shin, "Detecting energy-greedy anomalies and mobile malware variants," in *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services*. New York, NY, USA: ACM, 2008, pp. 239–252.

[3] G. Jacoby and N. Davis, "Battery-based intrusion detection," in *Global Telecommunications Conference, 2004. GLOBECOM '04. IEEE*, vol. 4, 2004, pp. 2250–2255.

[4] T. K. Buennemeyer, T. M. Nelson, L. M. Clagett, J. P. Dunning, R. C. Marchany, and J. G. Tront, "Mobile device profiling and intrusion detection using smart batteries," in *HICSS '08: Proceedings of the Proceedings of the 41st Annual Hawaii International Conference on System Sciences*. Washington, DC, USA: IEEE Computer Society, 2008, p. 296.

[5] D. Venugopal and G. Hu, "Efficient signature based malware detection on mobile devices," *Mobile Information Systems*, vol. 4, no. 1, pp. 33–49, 2008. [Online]. Available: http://iospress.metapress.com/content/w283084718113647

[6] J. Cheng, S. H. Y. Wong, H. Yang, and S. Lu, "Smartsiren: virus detection and alert for smartphones," in *International Conference on Mobile Systems, Applications, and Services (Mobisys 2007)*, 2007, pp. 258–271.

[7] D. Samfat and R. Molva, "IDAMN: An Intrusion Detection Architecture for Mobile Networks," *IEEE Journal on Selected Areas in Communications*, vol. 15, no. 7, pp. 1373–1380, Sep. 1997.

[8] M. Miettinen, P. Halonen, and K. Hätönen, "Host-Based Intrusion Detection for Advanced Mobile Devices," in *AINA '06: Proceedings of the 20th International Conference on Advanced Information*

Networking and Applications - Volume 2 (AINA'06). Washington, DC, USA: IEEE Computer Society, 2006, pp. 72–76.

[9] A. Bose, X. Hu, K. G. Shin, and T. Park, "Behavioral detection of malware on mobile handsets," in *Proceeding of the 6th international conference on Mobile systems, applications, and services*. Breckenridge, CO, USA: ACM, 2008, pp. 225–238.

[10] A.-D. Schmidt, F. Peters, F. Lamour, and S. Albayrak, "Monitoring smartphones for anomaly detection," in *MOBILWARE 2008, International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*, Innsbruck, Austria, 2008.

[11] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," *Proceedings of the 12th USENIX Security Symposium*, pp. 169–186, 2003.

[12] D. S. R. Q. Zhang, "Metaaware: Identifying metamorphic malware," *in Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2007.

[13] F. V. Christopher Kruegel, William Robertson and G. Vigna, "Static disassembly of obfuscated binaries," *USENIX Security Symposium*, vol. Volume 13, pp. 18 – 18, 2004.

[14] C. H. T. Wang, C. Wu, "A virus prevention model based on static analysis and data mining methods," *Computer and Information Technology Workshops*, pp. 288–293, 2008.

[15] S. J. S. Eleazar Eskin, Matthew G. Schultz and E. Zadok, "Data mining methods for detection of new malicious executables," *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, 2001.

[16] INSECURE.ORG, "Top 100 network security tools," 2006. [Online]. Available: http://sectools.org/

[17] R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1(1), pp. 81–106, 1986.