SAARLAND UNIVERSITY
Faculty of Natural Sciences and Technology I
Department of Computer Science

BACHELORS'S THESIS

Developing a RiskScore for Android Applications

submitted by

Marc Schweig

submitted October 01, 2013

Supervisor
Prof. Dr. Michael Backes
Dr. Matteo Maffei

Reviewer
Prof. Dr. Michael Backes
Dr. Matteo Maffei

Advisor
Philipp von Styp-Rekowsky, M.Sc.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, ______________________ ______________________

Datum/Date Unterschrift/Signature

Abstract

The success of the Android operating system and the increasing capabilities of mobile devices made the Android OS a prominent target for malware developers. Since the security systems provided by Google have turned out to be insufficient to protect the users against security and privacy threats, the security of a user's private information is endangered and solely depends on which applications he decides to install. As this thesis shows, all risk signals available to the user, like permissions or community ratings, either are proven to be futile for the determination of an applications risk or are ignored by the users. Therefore, a risk scoring algorithm is created that enables users to identify the threads an application poses for the security of their private data . The developed tool, called RiskScore, is based on the compiled dex bytecode and the requested permissions of an application and provides a more accurate measure than comparable risk assessment techniques.

Contents

Chapter 1

Introduction

In 2007, Google founded the Open Handsent Alliance (OHA) [3], a consortium of mobile operators, hard - and software companies and phone manufacturers, with the goal to advance open standards for mobile devices. In the same year the OHA unveiled Android, their open source software flagship, which is distributed under the Apache license [1]. As an operating system (OS) for mobile devices Android has directly competed with other operating systems like Appel's iOS, Nokia's Symbian or Microsoft's Windows Mobile. However, due to the cost-saving licensing model of Android it became fast very popular among manufacturers.

Over the last few years Android has become more and more prevalent and its market share has reached 74,4% in the first quarter of 2013 [26]. With about 900 million activations, this makes Android by far the most common mobile device OS available [27].

But not only the Apache license made Android that popular. Shortly after the release of Android, Google integrated the Android Market into OHA's operating system. As an easy and uncomplicated distribution platform, the Android Market has become the favored store among all the software stores for mobile devices. In 2012, Google combined Google Music, Google Movies, Google Books and the Android Market to create a joined service called Google Play [24]. Google stated in July 2013 that the number of available applications was about 700.000 but recent market statistics estimate this number to be higher and to be located at about 810.000 [23], [6]. In May 2013, the number of application downloads from Google Play exceeded 48 billion applications [43].

Besides Google Play, a wide range of third-party marketplaces like Amazon AppStore or Samsung Apps are available. This diversity redounds to the customers' as well as to the developers' advantage. Developers can choose the markets with the best conditions and customers can benefit from promotions. However, the system's popularity and diversity have some downsides. In addition to honest developers, attackers and malware developers have focused on the Android operating system and its users.

1.1 Motivation & Thesis Objective

Over the last years, smartphones have become more and more popular. Smartphones now account for 49.3% of overall sales of mobile devices, up from 44% in 4Q12 [5]. Due to increasing hardware capabilities, users have started to use their smartphones for tasks which formerly were exclusive to computers, like social networking, shopping and even online banking.

Therefore it is inevitable for users to recognize dangerous applications in order to avoid the theft of sensible information, especially if one considers Google's application vetting mechanism, called Bouncer. The Bouncer is an automated malware detection system, with the goal to prevent malware uploads and to remove malware from Google Play. It analyzes newly uploaded applications as well as already distributed applications and developer accounts. To do so the Bouncer does not only scan for known malware, spyware or trojan signatures, it also performs a behavioral analysis to detect malicious behavior [29]. However, the Bouncer is bypassable and easily deceivable and therefore insufficient if one wants to ensure end user security.

One of the first successful attacks on the Bouncer was performed by Jon Oberheide and Charlie Miller. They uploaded a malicious apk - file which handed them a connect-back shell. This shell enabled them to explore and fingerprint the Bouncer's environment [32], [33]. Another successful attack was launched by researchers from the Trustwave SpiderLabs [40]. They used a hybrid approach for application development by programming their application in html and javascript distributing it in a native wrapper. This approach allowed them to dynamically reload malicious functionality and to bypass updates via Google Play. The Bouncer regularly analyzed this application but was not successful classifying it as malware until the developers wanted it [31].

Besides its Bouncer, Google uses a system called mandatory access control securing the access of Android applications to resources. Applications have to request certain permissions to get access to security critical functionalities and user data. Despite the fact that this system gives information about potentially suspicious applications, research[1] has shown that most

[1]This paper addresses the Windows AUC notifications, however the results can easily be transferred to the Android environment

users simply ignore such requests and blindly grant all the rights an application requires [30].

Another mechanism Google based Android's security upon is a reporting system. Users can inform Google about malicious applications and in case Google is able to confirm the maliciousness, the concerning applications are removed from Google Play. Additionally, Google is able delete those applications from the users smartphones by triggering a kill switch [9]. But this only attenuates the effects instead of preventing the occurrence of zero-day malware.

Google omits a manual review process, in contrast to Apple and Microsoft, and this system has undeniable advantages, like a faster deployment of new applications and a more dynamic Google Play content. However, it is the dynamic application landscape in addition to the vast distribution and the insufficient security procedures that have made Android a remunerative target for malware developers, with the tendency to rise. Among all Android application markets, Google Play is still one of the most secure stores compared to other distribution platforms which nearly consist solely of malware [42].

In the beginning of 2013, BadNews became a significant development in the evolution of mobile malware, it achieved an enormous distribution within Google Play and remained undiscovered until it had been downloaded more than 2 million times [37]. But in consideration of the fact that none of the leading mobile anti-virus systems is able to detect malware that is unknown up to this day, this number could be much higher [28]. Statistics show that the amount of found malware samples has risen to 100.000 within the last year, most of them have been programmed with the goal to collect sensitive and private information [11].

Due to this radical development, users need a simple and comprehensible guideline to choose trustworthy applications and to reject potentially dangerous ones. A lot of solutions have been presented in the past, but previous work mainly focused on either permissions [21], [35], code [18], [45], [25], [44], [22] or runtime behavior [17] but rarely on combinations of them. This and other facts make them infeasible for the standard end user, since a permission focused approach will surely miss malicious applications specialised in root exploits and runtime monitoring as well as a heavy weight static analysis are too demanding to be practical.

The goal of this thesis is to develop a method to assess the risk of applications using more than one type of input, like e.g. function calls, permissions or metainformation. To be expedient by endusers, this method shall not require root privileges or drastically affect the runtime of rated applications and it should be capable of being integrated into security focused applications. Those challenges cause some limitations for this thesis. Therefore, the first task is to identify and to define meaningful and suitable risk signals, in other words risk signals which are easily approachable and still have significance. In a second step, those risk signals have to be extracted. For permissions this requires a manifest file parser and for function calls a dalvic bytecode assembler / disassembler like dexlib [4]. Another task is to develop a risk scoring function out of those risk signals. Therefore, it is necessary to classify risk signals such that the powers e.g. which are obtained by requesting a permission are incorporated within the representation of the risk signal. Those classified signals can then be used to assess the risk of an application requesting or possessing the corresponding risk signals. The generated score has to satisfy conditions, to be developed during this thesis, which enable users to conclude why an application scored the way it did.

Another thesis objective is the implementation of this risk scoring technique and the development of a tool, called RiskScore, to perform the risk estimation of Android applications. Therefore, RiskScore has to generate comprehensible output which constitutes a guideline for users utilizing this tool. Based on the results generated by this tool, the user then can decide whether or not he wants to monitor installed applications to let them perform unsupervised or to uninstall applications too risky to remain on the system.

1.2 Outline

The remaining thesis describes in detail the development of the approach used within the risk estimation framework as well as the framework itself. It is structured as follows:

Chapter 2 provides the necessary background information about Android to fully understand this thesis. It roughly describes the core system and important features. Furthermore, it gives a short overview of the runtime environment including the DVM, Android's virtual machine, before the bytecode - and application structure are described in detail, as they are fundamental for the understanding of the risk evaluation framework.

A high level overview of the risk scoring framework is given in *Chapter 3* of this thesis. This chapter also briefly describes the functionality and the task of the components used within the RiskScore.

The risk signal selection is depicted in *Chapter 4*. Before the design and development of the actual risk score are described in detail, this chapter outlines the evaluation and selection of the risk signals used in the risk evaluation framework. In addition, it gives a short overview of some exploits and attacks on the Android operating system.

As mentioned before, *Chapter 5* explains in detail the design and development of the risk assessment framework and represents besides *Chapter 4* the core part of this thesis. To generate comprehensive output for the user this chapter introduces the required desiderata for the risk score and depicts a method to initialize and categorize the risk signal data so that all needed information is incorporated within the risk signal representation. Afterwards, the risk scoring function is developed based on these fundamentals.

One of the goals of this thesis is to implement the theory described in the last chapter into a concrete tool. *Chapter 6* provides details about the implementation of the RiskScore. This comprises the use of all algorithms and third-party tools that are included in the RiskScore to implement certain tasks.

Chapter 7 describes the effectiveness of the risk evaluation framework. Therefore, the output of the RiskScore is analyzed in detail and afterwards compared to common risk assessment techniques.

The results and contributions of this thesis are described in *Chapter 8*.

Chapter 2

The Android Operating System

The Android operating system for mobile devices has originally been developed by Android, Inc., which has later been bought by Google in 2005. Two years later, Google founded the Open Handset Alliance [3] to continue the development of Android as a open mobile platform. At present, the OHA is a consortium of 84 companies from different areas, members are among others mobile operators, hard - and software companies as well as phone manufacturers. In the year of the foundation of the OHA, Google published in cooperation with the alliance the first version of the Android SDK.

About one year later, the first official Android software version 1.0 was published. In the same year, HTC presented the first smartphone running Android, the HTC Dream and most of the Android source code was made available, with the exception of some applications developed by Google. For the last five years, the OHA has constantly developed Android further and therefore, a multitude of updates has been released that have targeted either the operating system or applications shipped as part of Android. Those updates include bugfixes, integrate new features into the system or refine already integrated ones. Since version 1.5, the new Android versions have received a new codename based on the name of a dessert with every major update. Some of them are KitKat (since version 4.4), Jelly Bean (since version 4.1), Ice Cream Sandwich (since version 4.0), Honeycomb (since version 3.0) and Gingerbread (since version 2.3).

The Android software stack consists of several layers, namely a Linux kernel, a middleware which includes native libraries, the reference monitor and the runtime environment, and an application framework for the execution of applications. *Figure 2.1* displays a high-level overview of the Android architecture.

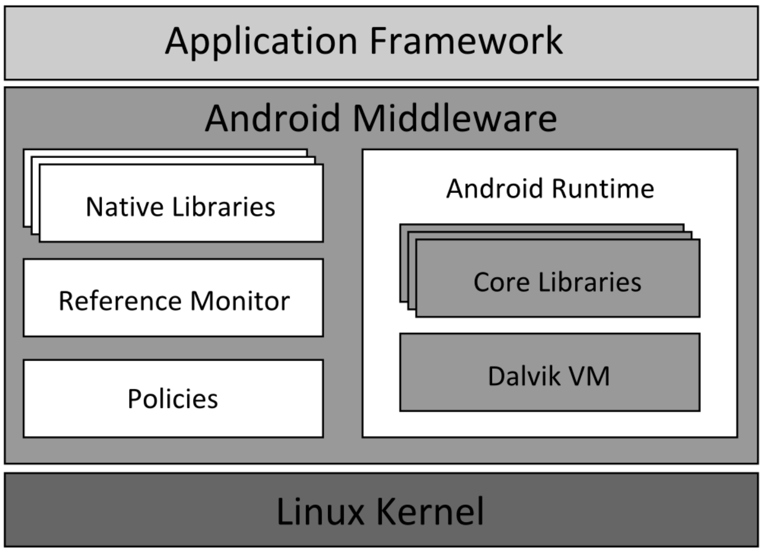

Figure 2.1: High-level overview of the Android architecture

2.1 Android Kernel & Middleware

Since the introduction of Ice Cream Sandwich, the customized Android kernel is based on the open source Linux kernel version 3.x. Therefore, Android itself does not have to provide or implement any core functionality, like process and thread management or memory management.

The security model of Android applications has been based upon a sandboxing mechanism implemented in the underlying Linux. Therefore, each application is executed with an own user ID (UID) within its own Dalvik virtual machine (DVM) which itself is again running in its own process. Thereby, only the Linux kernel can directly access the phone's hardware and applications can only access a very limited set of system features and their own data.

But to allow more functionality and more complex systems, it is necessary for an application to be able to interact with other applications or the hardware. For this reason, Android integrates the Binder, a customized implementation of the OpenBinder, into its kernel to enable controlled interprocess communication (IPC).

Figure 2.2 depicts the communication between applications via the Binder. To ensure that only authorized applications access protected data or functionalities, a reference monitor enforces a mandatory access control mechanism based on permission, which applications can request in their manifest file.

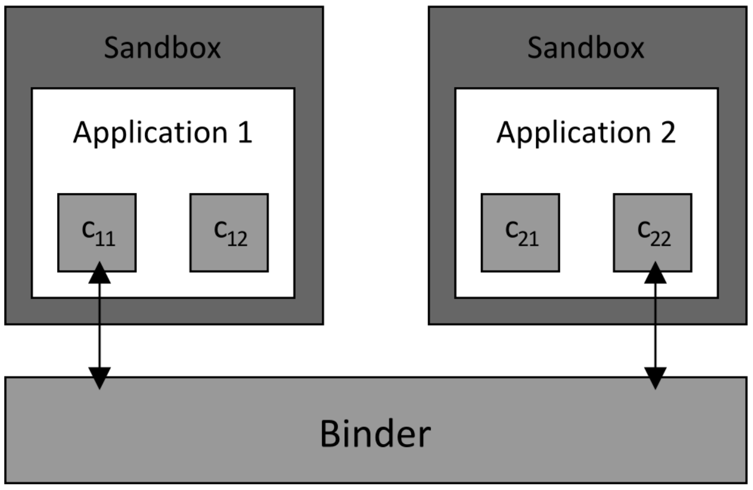

Figure 2.2: Inter process communication via the Binder

2.2 Android Runtime & Dalvik Virtual Machine

Considering the design choices made while developing the Android operating system, it becomes obvious why a custom VM has been implemented into the operating system and why the well proven Java VM is disadvantageous.

First of all, the ecosystem provides only limited resources, since Android has been developed as an OS for mobile devices. Therefore, the memory requirements of VM instances have to be kept minimal. Furthermore, smartphone users frequently switch from one application to another, causing applications to be opened and closed in a short period of time and lastly Android has to keep high-performance even if several Virtual Machine instances are running simultaneously. Therefore, Android implements its own virtual machine, called Dalvik or short DVM, which implements several optimizations matching the requirements of a mobile device.

As a result of these optimizations the DVM does not support Java bytecode anymore, but the Android SDK provides a tool, called dx, which translates Java class files into dalvic executable bytecode (dex bytecode).

In addition to the optimizations of the VM implemented in Android, it is possible to implement especially time-critical applications in C/C++ using the Native Development Kit provided by Google [12], since native code runs up to ten times faster than Java bytecode executed in a virtual machine. However, the DVM indeed spawns native code running in the same thread inheriting the UID and the permissions of the parent application, but the kernel's call interface is exposed to native code and that enables those applications to exploit kernel vulnerabilities and privileged daemons to "root" the system [44].

2.2.1 The dx compiler

The dx compiler is used to convert multiple Java class files containing Java bytecode into a single dex file, named `classes.dex`, containing the dex bytecode. This compiling consits of several steps, namely the translation, reconstruction and interpretation of the constant pools, the class definitions and the data segment.

The constant pool contains all the constants used by a class, like references to other classes, method names or numerical constants. Java applications duplicate those constants, since the Java bytecode is distributed among many class files. The dx compiler eliminates those replications, as the DVM uses a single pool that all classes simultaneously utilize. Furthermore, integers, long integers and single and double floating point elements are inlined directly into the dex bytecode [18].

The class definitions as well as the data segments are taken from the Java class files during the compilation and are rearranged within the dex file.

The concept of a shared constant pool and a single dex file has one major advantage for the application to mobile devices compared to the concept of multiple classes files. The final bytecode size is drastically reduced. *Table 2.1* displays a comparison of different bytecode sizes of common system libraries and applications integrated into the Android system [8].

Input	Jar file	Compressed jar file	Dex file
Common system libraries	100%	50%	48%
Web browser app	100%	49%	44%
Alarm clock app	100%	50%	44%

Table 2.1: Bytecode size comparison

2.2.2 The Zygote

Besides the significant size reduction of dex bytecode by the dx compiler, the Zygote is another optimization concept that targets the memory usage and reaction times of the virtual machine implemented in Android. Since there is no memory sharing between traditional Java VM instances, each instance loads a copy of core library classes and associated heap objects. Because of the additional workload, the cold startup, a startup with the additional workload to initialize core libraries, of those VMs takes a tremendous amount of time. But for mobile devices, where users frequently open and close applications this is not acceptable.

To speed up cold startups and to keep the memory footprint of virtual machine instances minimal Android applies a concept called Zygote. Under the assumption that core library classes and their corresponding heap objects are used frequently within different instances of the DVM, Zygote keeps them in a for read-only shared memory. In case an application needs to write to the shared memory, the concerning area is copied to the corresponding application's VM process.

Zygote is a VM process started at boot time, that itself spawns a DVM which pre-loads and pre-initializes common core library classes. Afterwards, it awaits requests from the runtime process to create child VMs for applications via an open socket.

2.2.3 Register-based Virtual Machine

The Dalvik Virtual Machine is register-based instead of stack-based, like traditional Java VMs. Java bytecode is able to assign local variables to a local variable table before they get pushed onto the operand stack. However, it is also possible to work directly on the stack without storing the variables in the local variable table. On this operand stack, the variables then can be manipulated by opcodes.

The DVM bytecode assigns its local variables to one of 2^{16} available registers instead. These registers can directly be manipulated by the opcodes.

Since the stack has to be kept in memory, stack-based virtual machines have a larger memory overhead and perform significantly weaker than register-based virtual machines, but they have the advantage of being easier to implement.

According to Shi et al. [39], register-based bytecode needs on average 47% fewer instructions than stack-based bytecode but has about 25% larger code size [8]. Benchmarks have shown that the average execution time of register-based bytecode is 32.3% faster than the execution time of the same program in stack-based code.

Keeping the significant code size reduction through the dex file format and the concept of a shared constant pool in mind, the gain of 25% code size because of the register-based instruction set is more than compensated, so that the Dalvik virtual machine still has a more efficient memory usage than a traditional stack-based virtual machine. Furthermore, the DVM can benefit from a faster bytecode execution which redounds to the users' and the developers' advantage, since mobile devices typically provide little computational power.

2.2.4 The Bytecode Structure of the DVM

The register-based architecture of the Dalvik VM makes the instruction set substantially different from the one of stack-based Java VMs. The dex bytecode currently consists of 218 different opcodes, whereas the Java bytecode only has 200. However, the purpose of these opcodes differs considerably. While Java implements tens of opcodes for the purpose of pushing elements between the stack and the local variable table, Dalvic opcodes have no need to provide such functionality, instead, they are dedicated to move content from one register to another, to invoke e.g. methods or to perform unary or binary operations directly on the stored values.

Dex bytecode includes instructions that not necessarily preserve the data type of their arguments, e.g. the instruction `'const vAA, #+BBBBBBBB'` does not specify the type of the 32-bit constant.

Furthermore, the instruction set has been optimized to reduce the number of needed instructions and registers and to further minimize the size of the dex file. Therefore, 2-address instructions for binary operations have been established. These operations are labeled with a trailing `2addr`. Here, the first specified register also determines the destination register, since the instruction format follows a strict argument ordering. Another optimization of the dex bytecode is the introduction of instructions that work with register/literal pairs as arguments. These instructions can be performed on 8-bit or 16-bit integer values and are marked with `lit8` and `lit16` respectively.

2.3 The Android Application Framework

Android applications typically consist of multiple components to facilitate reusability and to provide modularity. Components of an application have to be declared within the applications manifest file, like e.g. `Activities`, `Content Providers`, `Intents` or `Services`.

Although, all modules an application consists of have the commonality that they have to be declared in the `AndroidManifest.xml`, their actual functionalities may differ. Activities are one screen of the graphical user interface of an application, therefore an application usually consists of several different activities. On the other hand, a service is a background process running without user interaction and an application can be composed of multiple services or solely consist of one service [14].

At the application level, Android also enforces several security features, like application signing or its mandatory access control. Application signing ensures that only developers who have signed the application with a self-signed certificate ship updates for their applications. Besides the verification of the code origin, this feature enables data sharing between applications signed with the same key, since they are executed with the same `UID`.

The mandatory access control system that protects the sensitive user data and risky or potential dangerous functionality from misuse, is based on permissions. These permissions have to be defined in the applications' manifest file, similar to the components an application consists of. Although there exists a predefined permission set, developers have the ability to define their own permissions.

Each permission can be categorized based on the level of protection the rights have that this permission grants. There are four different protection levels, in ascending order of their level they are: *normal*, *dangerous*, *signature* and *signatureOrSystem* [15]. A permission label is assigned to a permission matching corresponding description:

normal: This permission type is automatically granted during the installation, since the contained permissions only grant access to isolated application-level features, with minimal risk. An example for a normal permission is the *VIBRATE* permission.

dangerous: A permission tagged dangerous gives the requesting application access to private user data or the ability to perform potentially dangerous activities. Therefore, these permissions have to be granted by a user before the application can be installed. An example for a dangerous permission is the *SEND_SMS* permission.

signature: Signed permissions are automatically granted by the system to requesting applications signed with the same certificate as the application that created that permission.

signatureOrSystem: Permissions that are granted by the system to applications that are part of the Android system image or that are signed with the same certificate as the declaring application. In general, this security level is only used by Google or vendors.

After the permissions are set at installation time, the Reference Monitor enforces the abidance during runtime. At the moment, there exists no convenient system feature that allows the user to dynamically revoke permission or to grant additional permission after the installation. However, in the Android version 4.3 a new activity has been introduced called *App Ops*. Due to the fact that this feature is still in development, this activity is not fully functional, but it enables the user to manage the permissions used by some applications. Furthermore, it is speculated that App Ops will become an inherent part of Android's system applications after the development is finished.

There are other possibilities for users to integrate a dynamic permission management into their Android running mobile device, however, almost every system requires root privileges to do so.

Chapter 3

RiskScore - A high-level overview

In general, it should be assumable by the user that applications, especially those obtained from a trusted source like Google Play, neither leak privacy sensitive information nor perform other malicious activities. But, in reality, this assumption turns out to be wrong, since even non-malicious applications transmit private information. In fact most freeware applications that include advertisement or analytic libraries tend to have privacy sinks implemented. This thesis endows users with the ability to protect their data by indicating the riskiness of applications. In addition to the development of an theoretical model, a concrete implementation of the risk evaluation framework is provided as part of this thesis. This chapter presents an high level abstraction of the evaluation framework, to be implemented.

Since the source code of Android applications is not available in general, the analysis of code based risk signals has to be performed on the applications bytecode. Android applications are programmed in Java and compiled to dex bytecode which gets executed in the Dalvik virtual machine. The bytecode, together with other resources like the manifest file, is distributed as an '.apk' - file which is also simply referred to as application by users. As the applications package contains the bytecode, it can easily be extracted. The same holds for the manifest file which contains the permission information needed for the analysis of the permission based risk signals.

The analysis performed by the RiskScore consists of multiple steps. First, the corresponding files, as there are the manifest and the classes.dex, get extracted and preprocessed and RiskScore parses the needed information from these files. Only if the application contains risk signals at all, the analysis continues with the execution of the actual risk ranking algorithm.

3.1 System Overview

Figure 3.1 shows a high-level abstraction of the risk evaluation framework. The framework actually consists of several interchangeable modules to increase flexibility and to ease the evaluation of different components. To estimate the risk of an application, the corresponding risk signals have to be parsed and preprocessed. This functionality is provided by the parser package.

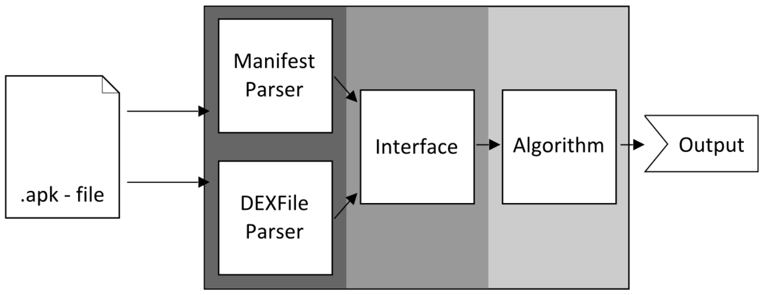

Figure 3.1: High-level view of the risk assessment framework.

Since the analysis of dex bytecode is impracticable and inconvenient, risk assessment tool bases the code analysis upon a more convenient assembly language. Therefore, the `DexFileParser` first preprocesses a given '.apk' - file and translates it before the set of risk signals gets extracted. However, the implementation of the disassembler is not part of this thesis, as there are some open-source disassemblers available which provide extensive functionality and generate a high quality assembly language.

Given that the manifest file is part of the '.apk' - file , the first preprocessing step of the `ManifestParser` is the extraction of the manifest. To solve this task, it again was possible to use a preimplemented open - source solution that processes the whole '.apk' - file and that outputs the permission set of an application as stated within the manifest - file.

After the preprocessing is finished, all data is handed to the `Interface` which passes it on to a data container called `Application`. Here, the risk signals and meta information of all previously described analysis steps are stored. In a subsequent step, the data container is processed by an instance of `Algorithm` whether as part of the initialization of the algorithm or in the course of the evaluation of the application.

3.2 Design Choices

The current design provides benefit beyond the objective of this thesis. It does not only provide a method to identify the threat an application poses that is based upon requested permissions, dex bytecode and meta information, it in fact provides a cyclopaedic framework for the risk assessment of all kinds of software.

The RiskScore implements three different parsers to collect the data needed for the application analysis. However, due to the modularity of the RiskScore, it is easily possible to interchange or to extend those components to augment the current algorithm or to reuse the provided algorithms for the analysis of other programs, e.g. the risk evaluation of chrome extensions.

Furthermore, it is easily possible to implement new algorithms for evaluation purposes and to integrate them into the system. The only thing necessary is to inherit from the base class `Algorithm` and to initialize the interface using the new instance of `Algorithm`.

Chapter 4

From Risk to a Risk Score

As an addition for security applications running on mobile devices the RiskScore should perform well even with the limited resources of smartphones. Therefore, the evaluation cannot be based on demanding and time consuming analyses like control flow graphs. On the other hand, the framework cannot require root privileges, since most users use unrooted Android versions. Because of those limitations the RiskScore will be based on a permission based approach [38].

The biggest drawback is that this system only considers an application's permission set as risk signals. However, the permission set is a very powerful indicator for the ability of an application to leak private information such as the contacts. Even though permission configurations can be used to fingerprint malware they do not suffice. There exists malware that cannot be detected using only an application's permission set. Root exploits, which clearly should be considered a risk, only require the ability to execute code and therefore no permission is needed.

To resolve this issue, the RiskScore will additionally consider other risk signals than permission sets of applications. Furthermore, the existing permission based indicators will be enhanced.

4.1 Code Based Risk Signals

To resolve the previously mentioned disadvantage of the model RiskScore is based upon, another set of risk signals is introduced. The ability to load and execute code. In [44] the occurrence probability of dynamically loaded code and java native code execution in applications is described as very low (about 0.58% for dynamic code loading and 4.52% for native code execution). This thesis is no able confirm those results. A in-depth analysis of the standard and the malware repository detected an integration rate of 2.9%

among benign applications and 19.6% among the malware samples.

Typically all Dalvic bytecode is contained within the `calsses.dex` file. But, using the `DexClassLoader` it is possible for applications to load classes from other files, e.g. `.jar` or `.apk`, even if they are stored on remote servers. This gives developers the power to radically change an applications behavior and to conditionally become malicious, e.g. after the approval of Google's application vetting system, the Bouncer. For this reason, applications which perform dynamic code loading expose users to greater risk than other applications without this ability.

Native code execution is more common than dynamic code loading but should not be considered less dangerous. Using JNI an Android application is able to execute code programmed in another language than Java. This exposes the kernel's call interface to those applications and enables them to exploit kernel vulnerabilities and privileged daemons to "root" the system [44]. To implement JNI it is necessary to either call `System.loadLibrary()`, `Runtime.getRuntime().loadLibrary()` or `System.load()` [41]. Therefore, it is easy to detect and can be penalized. The analysis of 20303 applications from the standard repository revealed that about 21% integrate JNI the same analysis applied on 1260 malware applications showed an integration rate of 54.12% for the malware repository.

At Black Hat USA 2012 [40] the Trustwave Spiderlabs presented "SMS Bloxor", an application which was able to circumvent Google's Bouncer by using native code execution of JavaScript code and dynamic code loading of Javascript code out of the JavaScript code.

Besides JNI, `Runtime.exec()` can be used to gain root priviliges on a mobile device running Android. DroidDream tries to achieve root priviliges using among other vulnerabilities the rageagainstthecage exploit which is based on the creation of many processes via `Runtime.exec()` [7], [44].

An analysis conducted as part of this thesis has revealed that about 50% of all malware applications use `Runtime.exec()` in comparison to about 21% of applications within the standard repository. An in-depth analysis has shown that about 52% of the malware applications containing `Runtime.exec()` included `Runtime.exec()` only because of ad - or analytic libraries. The same analysis depicted that about 67% of the including applications out of the standard repository included `Runtime.exec()` only because of ad - or analytic libraries. Therefore about 7% of the standard applications and about 24% of the malware samples included `Runtime.exec()` as part of their own functionality.

Therefore, dynamic code loading, native code execution and `Runtime.exec()` calls are established as risk signals within the proposed risk evaluation framework, because of their heightened occurrence within malware samples. The detailed results of this analysis can be learned from *Table 4.1*.

Risk signal	%benign	%malware
DC	2.90%	19.60%
JNI	21.57%	54.12%
Runtime.exec()	7.06%	24.04%

Table 4.1: JNI, dynamic code loading (DC) and Runtime.exec() rates

In course of this lightweight static analysis, it is possible to speculate about the misuse of sensitive and private information and therefore to enhance the permission based risk signals. As Enck et al. discovered in 'A Study of Android Application Security' ad - and analytic libraries are widespread within Android applications. About 51% of all applications include one or several ad - or analytic libraries and several of those libraries access sensitive information [18]. An analysis implemented affirmed the results of Enck et al. and extended them with more libraries. *Table 4.2* shows the implementation rate and the number of different permission guarded API calls for the five most common advertisement and analytic libraries.

Library package	Integration rate	Different API calls
com.google.ads.*	46.46%	5
.*analytics.*	14.36%	15
com.flurry.*	11.14%	5
com.millennialmedia.*	9.62%	6
com.bugsense.*	8.45%	9

Table 4.2: Ad- and analytic library statistics

Since the purpose of those libraries is well known, it is easy to speculate that permission protected API calls are data flow sources. For this reason, the use of such libraries will be a risk signal on its own if the application possesses one or more of the permissions requested by the integrated libraries.

4.2 Permission Based Risk Signals

Android's mandatory access control system and the resulting permissions are a good detection instrument of riskiness if one is concerned e.g. with information theft. Besides the possession of a specific permission which guards a security asset, the possession of certain permission configurations which are particularly dangerous will be considered by the RiskScore.

Enck et al. have developed a set of security rules which are basically combinations of permissions and which should not be possessed by an application [19]. Although, some of the security rules are outdated, since Google has updated the permission system, most of those rules still apply. *Table 4.3* shows the sanitized set of Kirin security rules.

Kirin Rule
SET_DEBUG_APP
READ_PHONE_STATE, RECORD_AUDIO, INTERNET
PROCESS_OUTGOING_CALLS, RECORD_AUDIO, INTERNET
ACCESS_FINE_LOCATION, INTERNET, RECEIVE_BOOT_COMPLETED
ACCESS_COARSE_LOCATION , INTERNET, RECEIVE_BOOT_COMPLETED
RECEIVE_SMS, WRITE_SMS
SEND_SMS, WRITE_SMS

Table 4.3: The set of the sanitized Kirin rules

Only 3.76% of all benign applications and 2.53 % of malware applications fulfilled the first rule. The corresponding rates for RECEIVE_SMS, WRITE_SMS and SEND_SMS, WRITE_SMS are 0.21% and 0.18% in the standard repository and 28.80% and 30.79% for the malware samples. All the other rules do not apply at all. In spite of the fact that the Kirin rules only infrequently correspond to a subset of the permissions of benign applications or malicious applications, those configurations are nevertheless a good indicator of riskiness. Matching applications still have the ability to cause harm despite their repository affiliation.

Another enhancement made is the analysis of malware configurations. During the analysis of 1260 different malware samples of 49 different malware families, it was possible to identify permission and code-based fingerprints for malware applications. Those fingerprints correlate in some parts with the Kirin rules and in other parts they complement it. The only identifiable fingerprint that is not already a Kirin rules and has significance is the combination of `READ_SMS` and `WRITE_SMS`. 50.31% of all malware samples implement this rule in contrast to only 0.21% of all applications in the standard repository.

4.3 Other Risk Signals

To augment the effectiveness and to compensate weaknesses of the model which underlies the RiskScore, a variety of risk signals have been considered besides permission-based and code-based risk signals. Other risk signals are mainly community-based and have been extracted from the PlayStore. However, this community-based information, which has been examined as potential risk signals in previous studies, have been determined to be ineffective [10].

Since the number of permissions is a basic risk score, it is possible to judge the usability of community-based meta information as a risk signal by observing the correlation of the potential signal with the number of requested permissions. Meta information is disqualified as a risk signal iff inputs with positive meaning, like the average star rating of an application, correlate positively with the number of permissions and vice versa. *Figure 4.1* and *Figure 4.2* depict the described correlations.

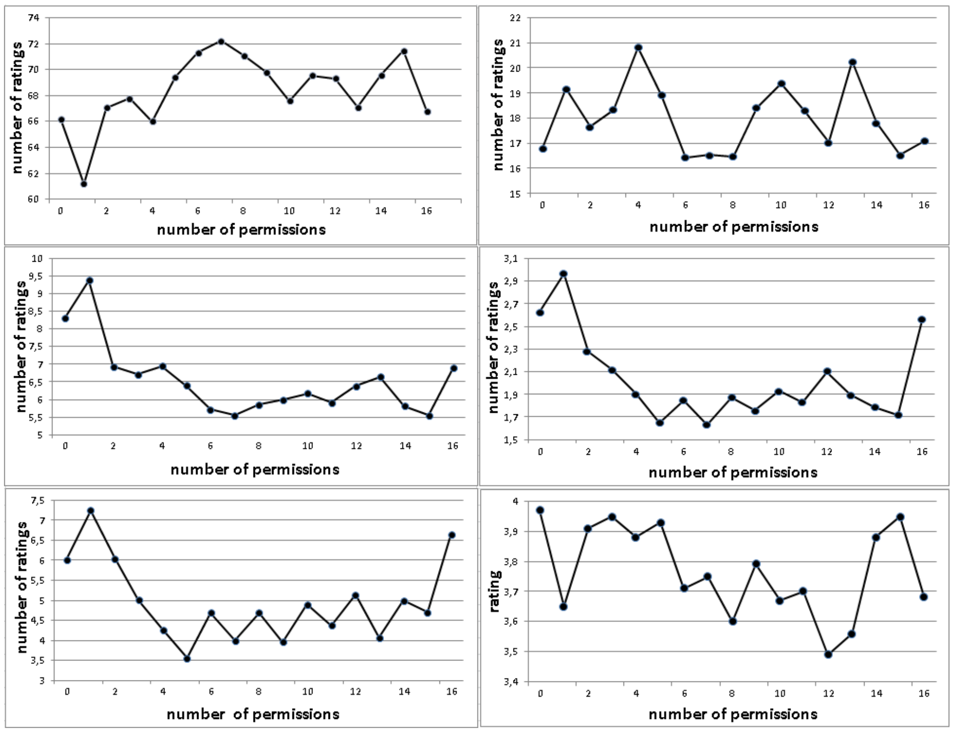

Figure 4.1: Correlation of the amount of five star, four star, three star, two star one star rating and the average rating with the number of permissions. (From top left to bottom right)

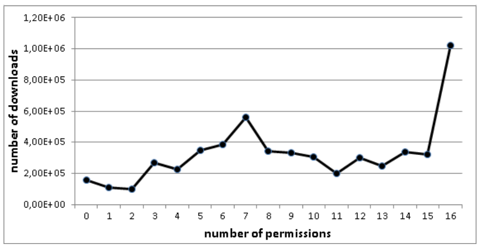

Figure 4.2: Correlation of the number of downloads with the number of permissions.

This thesis confirms the findings of [10] which describes the applications popularity and the community rating as unqualified because both are signals with positive meaning and both positively correlate with the number of requested permissions. The other community based signals as there are among else the number of downloads and the single star ratings are disqualified for the same reason.

Chapter 5

Bayesian Risk Assessment

As mentioned in the previous chapters, the implementation of the RiskScore is based upon the work of Peng et al.. The following section will introduce the work conducted in [35] and lays the foundation for the extension and the further development of this approach, which will be described in the subsequent section.

5.1 Naive Bayes with Informative Priors (PNB)

This thesis aims at advancing the work of Peng et al. such that two of their desiderata are fulfilled. The fulfillment the third desideratum, namely that the risk scoring function shall be easy to explain, does not improve the risk evaluation framework, therefore it has been decided not to insist on simplicity if it is not beneficial for the RisScore. The two remaining constraints are:

1. A risk function is monotonic.
2. Malicious applications generally have a high risk score.

In general, the i'th application in the dataset can be represented by the set of its applications or more formally $a_i = x_i = [x_{i,1}, \dots, x_{i,M}]$, where M is the number of permissions. Now, the fist desideratum can be formalized by the following definition.

Definition 1. *A risk scoring function rscore* : $\{0,1\}^M \to \mathbb{R}$ *is* ***monotonic*** *if and only if for all* $x_i, x_j \in \{0,1\}$ *holds that*

$$\begin{aligned} \exists k(x_{i,k} = 0 \land x_{j,k} = 1 \land \forall m(m \neq k \Rightarrow x_{i,m} = x_{j,m})) \\ \Rightarrow f(x_i) \leq f(x_j) \end{aligned}$$

The most promising probabilistic generative model for risk scoring is the naive Bayes with informative priors, since all other introduced models either are less accurate or lose the monotonicity constraint [35].

The PNB defines the risk scoring function as a monotonically decreasing function with respect to probability of an application being generated, so that a lower probability means a higher risk score. For example, using $rscore(a_i) = -ln(p(a_i|\theta))$ satisfies the condition.

The risk scoring function can be defined as $rscore : \{0,1\}^M \rightarrow \mathbb{R}$. The concrete $rscore(x_i)$ of an application x_i can now be computed by M assumed to be independent Bernoulli random variables

$$p(x_i) = \prod_{m=1}^{M} p(x_{i,m}) = \prod_{m=1}^{M} \theta_m^{x_{i,m}} (1-\theta_m)^{(1-x_{i,m})}$$

where $\theta_m \equiv p(x_{i,m} = 1)$ is the Bernoulli parameter. Using a Beta prior $Beta(\theta_m|a_m, b_m)$ over each Parameter θ_m, to avoid overfitting within the estimation and to fit the model to noise, it is possible to determine the Maximum a posteriori (MAP) estimation

$$\hat{\theta}_m = \frac{\sum_{i=1}^{N} x_{i,m} + a_m}{N + a_m + b_m}$$

where N is the total number of applications for this model estimation. Since the informative priors differentiate permission categories one assigns, depending on the risk level of an permission, different values a_m and b_m. A possible differentiation of risk signals is described in [38], this selection can still be subclassified in risky and high risk permission with the high risk category consisting of a manually selected subset of the described selection. For most risky permissions $a_m = 1$, $b_m = 2N$, for risky permissions $a_m = 1$, $b_m = N$ and for all the other permissions $a_m = 1$ and $b_m = 1$. But, the categorization can easily be extended to more categories. This approach is widely used in Naive Bayes models to implement knowledge which is not part of the data set. *Table 5.1* shows the permissions classified as dangerous and most dangerous.

Permission	Category
ACCESS_COARSE_LOCATION	Most Dangerous
ACCESS_FINE_LOCATION	
PROCESS_OUTGOING_CALLS	
CALL_PHONE	
READ_CONTACTS	
WRITE_CONTACTS	
READ_SMS	
SEND_SMS	
INSTALL_PACKAGES	
READ_CALENDAR	Dangerous
READ_HISTORY_BOOKMARKS	
READ_PHONE_STATE	
RECEIVE_MMS	
RECEIVE_SMS	
RECORD_AUDIO)	
RECEIVE_WAP_PUSH	
READ_LOGS	
INTERNET	
MOUNT_UNMOUNT_FILESYSTEM	
WRITE_CALENDAR	
WRITE_HISTORY_BOOKMARKS	
WRITE_SMS	
WRITE_EXTERNAL_STORAGE	
NFC	
GET_ACCOUNTS	
BLUETOOTH	
BLUETOOTH_ADMIN	

Table 5.1: Classification of Android permissions

However, the priors are not limited to the passing of additional information, it can also be used to ensure the fulfillment of the monotonicity property. It is obvious that every risk scoring function achieves this condition if $\forall m \in \{1, \ldots, M\}$ $\theta_m < 0.5$ holds, since changing any $x_{i,m}$ from 0 to 1 changes the probability by a factor of $\frac{\theta_m}{1-\theta_m}$, which is less than 1 if and only if $\theta_m < 0.5$. Therefore, the monotonicity property will always hold if one chooses the right prior.

An analysis conducted as part of this thesis indicates, most malware samples request more permissions than most applications out of the standard repository, these findings confirm the findings of Peng et al.. On average, applications from the malware repository requested 9.91 permissions and standard repository applications requested 6.53 permissions. This implies that the ranking of malware applications will be higher than the ranking of standard repository applications. Therefore, the second desideratum is fulfilled, too.

5.2 Enhancing the PNB (ePNB)

To augment the PNB it is inevitable to penalize applications if a subset of their permission sets matches at least one of the sanitized Kirin rules or malware fingerprints. To implement this, a penalty factor $\frac{1}{\gamma^r}$ is added where r is the number of matching rules and fingerprints. An extensive search revealed that $\gamma = N$ provides the best results.

Furthermore, the detection of native code execution, dynamic code loading and `Runtime.exec()` is an important part of this thesis, because those functionalities are among the most dangerous ones and can be used to execute root exploits or to conceal malicious activities. It is reasonable to constitute them as own permissions, as they permit those applications that implement them to perform possibly dangerous activities which would not have been possible before.

Since the integration of ad - or analytic libraries that request a permission into applications that possesses the corresponding permission enables to deduce that most likely the guarded information will be leaked, it is necessary to penalize such configurations appropriately. The best way to realize this is to incorporate the semantic information into the prior distributions and to add a new level of riskiness above most risky. If now guarded API calls exist within ad - or analytic libraries which are integrated into applications that possess the needed permission to perform this call, the corresponding permission will be raised to the next level.

The ePNB still fulfills the monotonicity property.

Proof. Changing a permission $x_{i,m}$ from 0 to 1 has more extensive effects now and to prove monotonicity it is necessary to distinct four different cases. The added risk signal could cause an application to fulfill a Kirin rule or a malware fingerprint (1), the risk signal could affect another risk signal(2), none of the cases mentioned before could apply (3) or both of them could apply (3).

(1) If this is the case, then the risk score changes by a factor of $\frac{\theta_m}{N(1-\theta_m)}$. $\frac{1}{N} < 1$ obviously holds for all $N > 1$ and the value of $\frac{\theta_m}{1-\theta_m}$ is discussed in part (3) of the proof.

(2) This can only occur if the application integrates an ad- or analytic library or if the changed signal 'integrates' such an library into the application. Either way the score changes by a factor less or equal to one. If the application possesses a permission that can be exploited by the newly added library signal or if the application already possesses the library signal and the application gains the permission that corresponds to a already possessed library signal the score changes by a factor $a < 1$. If the application does not own the permission associated with the added library signal, then the score does not change.

(3) As section 5.1 shows, every newly introduced risk signal is associated with a value $\theta_m < 0.5$ and therefore, the possession of any $x_{i,m}$ changes the score by a factor of $\frac{\theta_m}{1-\theta_m} < 1$.

(4) Concluding from (1), (2) and (3) it holds that if both cases co-occur, the resulting risk score decreases too.

Since for all cases the risk score either decreases or remains the same, the ePNB still fulfills the monotonicity property. □

The second desideratum can be proven by arguing that the occurrence probability of all risk signals is significantly higher if one considers malware applications. This has been shown in the sections 4 and 5.

Chapter 6

Implementation

After the theoretical development of the risk evaluation framework has been conducted during the last chapter, the following sections are devoted to the description of the implementation of the concrete tool called RiskScore. The tool itself is programmed in Java and can, due to its modular design, either be used as a standalone application or be integrated into security focused applications. *Figure 6.1* depicts a high-level overview of the RiskScore.

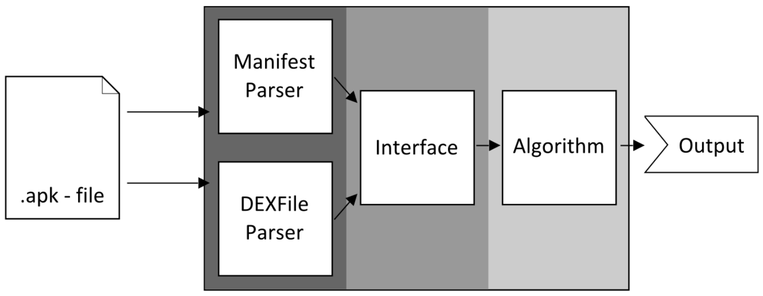

Figure 6.1: High-level view of the risk assessment framework.

As the abstraction shows, the RiskScores components can roughly be divided into three groups of different functionality. Each group will be explained in detail in the following sections.

Before the actual techniques responsible for data parsing can be explained, it is essential to at least sketchily understand the structure of the application repository used in this thesis, since the design choices of the implementation are affected by it. The application repository consists of the data of 20302 different applications. Thereby, each application packed as an '.apk' file is stored in a directory matching the applications name. The malware samples are contains in a different repository that contains 1260 different '.apk' files.

6.1 Application Package Parsing

To extract the needed permission information as well as the desired code-based risk signals, it is necessary to process the apk package of an application, since it contains the *'classes.dex'* as well as the *'AndroidMaifest.xml'*. Therefore both, the `ManifestParser` instance and the `DEXFileParser` instance work on the apk package after their creation.

6.1.1 Manifest File Parsing

The `ManifestParser` uses a `JarFile` to extract the *AndroidMaifest.xml* from the application package. After that, the manifest file gets decompressed and the relevant information gets stored into a `Set`.

Within an application's manifest file several elements can appear, however only the `<manifest>` and the `<appliaction>` element are required. Among elements of the same level no particular ordering is needed, except for the `<activity-alias>` element that always has to follow the `<activity>` element it is an alias for. Furthermore, formally, it is not necessary to assign attributes to elements, but in order to be adequately usable, some of them are nevertheless required, e.g. to declare the permissions used. Most of the assignable attributes start with an `'android:'` prefix, the only exceptions are some attributes of the root element [13].

It is obvious that the only relevant information are the `<permission>` elements and their associated `'android:name'` attributes. Since all permissions, even those declared by developers, are contained, it is still inevitable to sanitize the extracted permission set. However, this can be easily accomplished by composing the intersection between the extracted risk signal set and the set of predefined Android permissions. Therefore, each permission is identified by its stated `'android:name'`.

6.1.2 Dex File Parsing

The `DEXFileParser` integrates a disassembler to translate the *'classes.dex'* file into an more convenient assembly language. The RiskScore itself includes the *dexlib*, that is part of the baksmali tool, to read all needed code based information of an application [4]. However, there exist alternatives, like dexdump, the default disassembler integrated in the Android SDK [16] or the dedexer [34].

But in comparison, Baksmali offers a better and more structured code base than the dedexer which simplified the integration into the `DEXFileParser` and dexdump was disqualified for this task, since its output is harder to process.

After the creation of a `DexFile` instance, the whole classes.dex file is read and stored in different objects containing the data. The relevant data containers for this thesis are:

CodeItem: A `CodeItem` contains several `Instrcutions` and can be explicitly matched to one method containing it. The method's ID is also stored in a field of the `CodeItem`.

Instruction: An `Instruction` resembles a human readable mnemonic of an opcode in the dex bytecode.

InvokeInstruction: Some `Instructions` can also be represented as an `InvokeInstruction`, but only if it is an instruction that actually calls a method.

Figure 6.2 shows a code example of a simple HelloWorld program [4]. To understand it, it is necessary to get to know the used representation of *primitive types*, *reference types*, *methods* and *fields*.

```
.class public LHelloWorld;
.super Ljava/lang/Object;
.method public static main([Ljava/lang/String;)V.registers 2
sget-object v0, Ljava/lang/System;->out:Ljava/io/PrintStream;
const-string v1, "Hello World!"
invoke-virtual v0, v1, Ljava/io/PrintStream;->println(Ljava/lang/String;)V
return-void
.end method
```

Figure 6.2: A simple smali code example.

Primitive types are either void ('V'), boolean ('Z'), bate ('B'), short ('S'), vhar ('C'), int ('I'), long ('J'), float ('F') or double ('D'). Objects take the form of 'Lpackage/ObjectName;' where the 'L' indicates that this is an object type. Arrays can be identified because of a '[' in front of the type of the array, for more dimensional arrays the '[' has to be repeated.

'Lpackage/ObjectName;->MethodName(III)Z' is the representation of a Java method. Lpackage/ObjectName identifies the class that contains the method and MethodName refers to the name of the method. Furthermore, (III)Z is the method's signature with the unseperated parameters III (here, three ints) and Z is the return type (here, boolean).

Lastly, fields are diplayed similar to methods, first of all the containing class is identified and then the field itself by 'FieldName:Type'.

The `DEXFileParser` extracts two different information out of the applications bytecode. Therefore, the implemented algorithms search the bytecode for information about function calls out of ad- or analytic libraries that are accessing private information and potentially dangerous function calls that allow the execution of native code, enable to dynamically load code and calls to `Runtime.exec()`.

The first step of the extraction is to identify interesting code sections, namely those belonging to ad- or analytics libraries, since it is possible to speculate about their functionality and for that reason to infer if an accessed permission guarded API-call is legitimate. It is not legitimate if any data is accessed that reveals private information. Hence, all the `CodeItems` are checked if they are part of a method that is part of an ad- or analytic library and if they belong to such a library it is verified if they contained instructions that are associated with an illegitimate function call. Function calls that match these criteria are added to the set of found risk signals.

To filter these libraries a package blacklist is used similar to the one created by Enck et al. but the here integrated blacklist is more extensive [18], [36]. Furthermore, the association of instructions with permission calls is based upon work of Felt et al. [20].

The second part of the dex file parsing aims at identifying potentially dangerous function calls implemented by the scanned application. To do so, the parser verifies for all `Instructions` if they are `InvokeInstructions` and if they are, it is checked whether or not they correspond to blacklisted instruction. If they do they are added to the set of risk signals.

6.2 The Application Data Container

`Applications` are a data container for the extracted risk signals. Therefore, an `Application` object only contains the set of risk signals, besides a unique ID that matches the absolute path of the associated application.

6.3 The Interface

The `Interface` is responsible for the communication between the `Algorithm` and the parsers. It is used for the initialization of the algorithms as well as for the risk assessment of applications.

To instance an interface object, the algorithm used for the risk evaluation has to be assigned to the constructor, after that it is possible to initialize the algorithm object by handing it a set of `Application` instances or if the possibility is provided that the algorithm instance is able to discretely access a data set the corresponding method can be set off.

The application object is created by joining the information provided by the two used parsers. Therefore, each parser gets initialized with the corresponding path of an application. After that, the `ManifestParser` generates a set containing all permissions demanded by the application and the `DEXFileParser` provides all the code-based risk signal data in a set. Since every parser outputs a set of `Strings`, this information can easiliy be merged into a new set comprising all the risk signals.

If an `Algorithm` object shall be initialized with a set of `Applications` this procedure has to be repeated and the generated `Application` instances have to be stored into a set. But, if only this risk of the parsed application has to be computed, solely the `Application` is handed to the algorithm.

In case, it is necessary to adapt the RiskScore to the evaluation of e.g. Chrome extensions, this is the `Class` that needs to be changed. The parsers generating the new risk signals have to be inserted here replacing the old ones. It is also possible to integrate a new evaluation algorithm, but since the new algorithm can be handed at initialization time of the `Interface` object, the class does not need to be changed at all.

6.4 The Algorithm

Algorithm is only the `abstract` base class of an risk assessment algorithm. Every algorithm used in this tool has to inherit from `Algorithm` to be usable by an `Interface` instance, since the abstract class declares the methods an `Interface` object addresses.

6.4.1 Risk Assessment using The BayesianAlgorithm

The `BayesianAlgorithm` class is a concrete implementation of the theoretical technique developed in *Chapter 5*. It uses a data set that has been generated beforehand to speed up the instancing of this class. This precomputation is beneficial for the whole framework, since the assembling of a data base large enough to be significant is not feasible on mobile devices, the required amount of data and computational power would simply be too high.

The data base uses a JSON-like representation and consists of colon (':') separated name/value pairs which themselves are separated by a newline character. The slightly different structure has been chosen to make use of the built-in feature of the used `BufferedReader`, it reads files linewise.

Therefore, the choice of the `BufferedReader` is also advantageous for the overall running time of the risk evaluation framework.

The risk prediction of an application consists of two consecutive steps. First, the needed database gets imported.

After that, the risk signal set is matched against the malware fingerprints and Kirin rules, and an integer counter is incremented each time the risk signal set corresponds to one of the rules or fingerprints. At the same time, for each permission, its penalty class is determined. Therefore, the risk signal set contains the penalized permissions a second time with a prepended 'L' if ad- or analytic libraries have made use of them. For every permission that is identified as a higher risk permission that way, the penalty level is raised.

The risk assessment gets finished by multiplying all risk values of the signals together.

Chapter 7

Evaluation

To evaluate the RiskScore it is necessary to specify a sign of quality that the tool has to fulfill. Here, the measure of efficiency is based upon the second desideratum, namely that malicious applications generally have a high risk score, so that the generated score can be used to classify applications. Although, this measure does not fully correspond to all the qualities demanded from a risk score, like the presentation of a risk value in contrast to malware identification, it is the most common measure used to evaluate risk assessment tools and therefore it allows to compare the results of this analysis to those of other tools developed, e.g. [38], [35].

Besides a high risk score for malware applications, it is required that benign applications result in a significantly smaller score. Together, both requirements make it possible to differentiate benign and malicious applications.

7.1 Detailed Experimental Results

The RiskScore evaluation uses a 10-fold cross validation. Therefore, 10 subsets of the standard repository have been created. One subset was used as the training set of the risk evaluation tool and the other 9 datasets were tested by the framework. In a second step, the malware repository was analyzed by the RiskScore using each generated training set.

Figure 7.1 shows the experimental results that were generated by the evaluation of the PNB and the ePNB algorithm. At the first sight, the diagrams display similar results for both risk score functions. However, if the corresponding right-positive and false-positive rates are analyzed in detail, the ePNB algorithm performs better than the PNB algorithm.

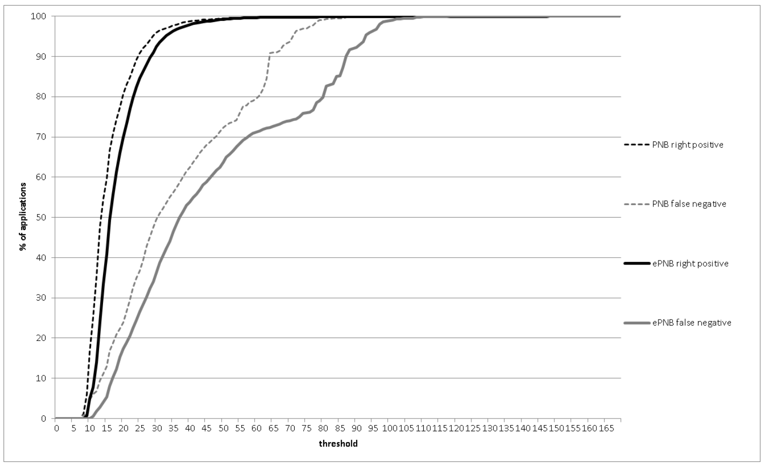

Figure 7.1: Comparison of the risk scoring algorithms

The ePNB more precisely differentiates between malicious applications and benign applications. If a malware detection rate of 81.22% is desired, the PNB falsely classifies about 29.22% of the benign applications as malware. However, the ePNB algorithm achieves at a detection rate of 80.32% a lower false-negative rate of about 24.01%.

7.2 Comparison to Other Tools

As pointed out before, the ePNB clearly outperforms the PNB. However, over the time a variety of tools for risk assessment have been created. This section discusses the results of the ePNB algorithm compared to Kirin [19] and a SVM classifier [38] which are among the most famous techniques for risk detection.

Since Kirin is only based on the fulfillment of its rules, it is not feasible to generate to depict the results of the Kirin rules in a figure. The Kirin rules applied on the malware dataset and PlayStore repository result in a detection rate of about 52.61% and a false-negative rate of about 4.02%. This is better than the result achieved by the ePNB, if a false-negative rate of less than 4% is demanded. However, the detection rate of only about 52.61% is to low and since the Kirin rule based detection is not adaptable, the ePNB is far more suitable as a risk scoring algorithm.

The risk scoring technique described in [38] uses a weighted SVM classifier with RBF Kernel to detect malicious applications. The detection rate of this approach is settled at about 80.99% whereas the false-negative rate only is about 8.12% at the same time. This ratio clearly outperforms the ePNB approach which at best achieves a right-negative/false-negative ratio of 79.32%/24.01%. But, the SVM classifier is only able to generate binary

(1 for benign applications and -1 for malicious applications) results. This, however, is not a desired output of a risk score, since it does not present a measure of the ability of an application to perform malicious activities. However, this thesis aims at creating a score that also presents minor risks to the user, instead of simply categorizing applications.

Chapter 8

Conclusion

This thesis provides a fast and reliable framework for risk assessment of Android application. However, the implementation of the risk scoring algorithm is not limited to this function, it is easily adaptable to the task of analyzing other software, e.g. Chrome extensions. In contrast to other approaches in this area, the RiskScore uses multiple risk signal sources to enhance its accuracy and still remains efficient considering its running time.

Therefore the selection of suitable risk signals was an essential part of this thesis. *Chapter 4* displayed the variety of risk signals and evaluated their convenience. Besides the requested permissions of an application, code based risk signals and community based risk signals have been considered. However, this thesis was able to prove the impracticality of community based signals. But, since the riskiness of an application is a measure of an application's ability to perform dangerous activities, it is necessary for the risk signal set to additionally comprise code based information, because there exist dangerous functionalities, like e.g. root exploits, do not necessarily require permissions.

The RiskScore itself benefits from the diversity of risk signals. The implemented framework outperforms other comparable risk assessment tools, since it is the only tool that generates a simple and comprehensible output and that is not restricted to malware detection. On the contrary, RiskScore generates output that enables to subclassify the riskiness of an application at will. Especially users benefit from this property of the risk score, since it enables them to define their own threshold value that an application's score is not allowed to exceed. Therefore the risk score helps users to secure their mobile device at a rate that suits them best.

Bibliography

[1] Android overview. http://www.openhandsetalliance.com/android_overview.html.

[2] Bytecode for the dalvik vm. http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html.

[3] Open handset alliance. http://www.openhandsetalliance.com/index.html.

[4] smali. an assembler/disassembler for android's dex format. https://code.google.com/p/smali/source/browse/dexlib/src/main/java/org/jf/dexlib/?r=a43de2411e7d8df902819554b21a273b58828d0a.

[5] Roberta Cozza Carolina Milanesi Annette Zimmermann CK Lu Tuong Huy Nguyen Sandy Shen Hugues J. De La Vergne Atsuro Sato Anshul Gupta, David Glenn. Market share: Mobile phones by region and country. http://www.gartner.com/id=2482417.

[6] AppBrain. Number of available android applications. http://www.appbrain.com/stats/number-of-android-apps.

[7] Axelle Apvrille. Android droiddream uses two vulnerabilities. http://blog.fortinet.com/android-droiddream-uses-two-vulnerabilities/.

[8] Dan Bornstein. Dalvik virtual machine internals. Google I/O 2008, 2008.

[9] Tim Bray. Exercising our remote application removal feature. http://android-developers.blogspot.de/2010/06/exercising-our-remote-application.html.

[10] Pern Hui Chia, Yusuke Yamamoto, and N. Asokan. Is this app safe?: a large scale study on application permissions and risk signals. In *Proceedings of the 21st international conference on World Wide Web*, pages 311–320. ACM, 2012.

[11] Denis Maslennikov Christian Funk. Obad - the most sophisticated android trojan ever. https://www.securelist.com/en/analysis/204792299/IT_Threat_ Evolution_Q2_2013#16.

[12] Android developers. Android ndk. http://developer.android.com/tools/sdk/ndk/index.html.

[13] Android developers. The androidmanifest.xml file. http://developer.android.com/guide/ topics/manifest/manifest-intro.html.

[14] Android developers. Package index. http://developer.android.com/reference/.

[15] Android developers. permission. http://developer.android.com/guide/ topics/manifest/permission-element.html.

[16] Android developers. Tools help. http://developer.android.com/tools/ help/index.html.

[17] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–6. USENIX Association, 2010.

[18] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A study of android application security. In *Proceedings of the 20th USENIX conference on Security*, pages 21–21. USENIX Association, 2011.

[19] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 235–245. ACM, 2009.

[20] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 627–638. ACM, 2011.

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

[22] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. Androidleaks: automatically detecting potential privacy leaks in android applications on a large scale. In *Proceedings of the 5th international conference on Trust and Trustworthy Computing*, pages 291–307. Springer-Verlag, 2012.

[23] Google. Apps - google play. https://play.google.com/about/apps/.

[24] Google. Google play. https://play.google.com/store.

[25] Michael C. Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *MobiSys*, pages 281–294. ACM, 2012.

[26] Rob van der Meulen Janessa Rivera. Gartner says asia/pacific led worldwide mobile phone sales to growth in first quarter of 2013. http://www.gartner.com/newsroom/id/2482816.

[27] Rachel King. Google i/o by the numbers: 900 million android activations. http://www.zdnet.com/google-io-by-the-numbers-900-million-android-activations-7000015432/.

[28] Shen Lin and Brian W. Kernighan. An effective heuristic algorithm for the travelling-salesman problem. Operations Research.

[29] Hiroshi Lockheimer. Android and security. http://googlemobile.blogspot.de/2012/02/android-and-security.html.

[30] Sara Motiee, Kirstie Hawkey, and Konstantin Beznosov. Do windows users follow the principle of least privilege?: investigating user account control practices. In *SOUPS*. ACM, 2010.

[31] Sean Schulte Nicholas J. Percoco. Failures of automated malware detection within mobile application markets. https://media.blackhat.com/bh-us-12/Briefings/Percoco/BH_US_12_Percoco_Adventures_in_Bouncerland_WP.pdf.

[32] Jon Oberheide. Dissecting android's bouncer. https://blog.duosecurity.com/2012/06/dissecting-androids-bouncer/.

[33] Jon Oberheide. Dissecting the android bouncer. http://jon.oberheide.org/blog/2012/06/21/dissecting-the-android-bouncer/.

[34] Gabor Paller. Understanding the dalvik bytecode with the dedexer tool. http://pallergabor.uw.hu/common/understandingdalvikbytecode.pdf.

[35] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 241–252. ACM, 2012.

[36] Artëm Perlov. Saarland University, Department of Computer Science, 2013.

[37] Marc Rogers. The bearer of badnews. https://blog.lookout.com/blog/2013/04/19/the-bearer-of-badnews-malware-google-play/.

[38] Bhaskar Pratim Sarma, Ninghui Li, Christopher S. Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: a perspective combining risks and benefits. In *SACMAT*, pages 13–22. ACM, 2012.

[39] Yunhe Shi, David Gregg, Andrew Beatty, and M. Anton Ertl. Virtual machine showdown: stack versus registers. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 153–163. ACM, 2005.

[40] Trustwave. Pinpoint your vulnerabilities. protect your business. https://www.trustwave.com/spiderlabs/.

[41] Christian Ullenboom. Eine c-funktion in ein java-programm einbinden. http://openbook.galileocomputing.de/java7/1507_21_002.html#dodtp8235387c-0f4e-45bf-9856-d4a0082e4a2c.

[42] Timothy Vidas and Nicolas Christin. Sweetening android lemon markets: measuring and combating malware in application marketplaces. In *CODASPY*, pages 197–208. ACM, 2013.

[43] Brad Ward. Google: 900 million android activations, 48 billion app installs. http://www.androidauthority.com/google-io-android-activations-210036/.

[44] Wu Zhou Xuxian Jiang Yajin Zhou, Zhi Wang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.

[45] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, pages 95–109. IEEE Computer Society, 2012.