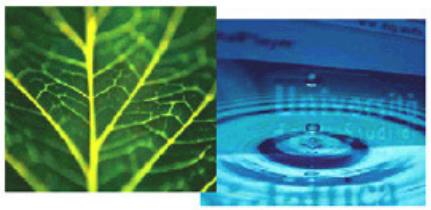
PhD Dissertation



International Doctorate School in Information and Communication Technologies

DIT - University of Trento

Improving the security of the Android ecosystem

Yury Zhauniarovich

Advisor:

Prof. Bruno Crispo

Università degli Studi di Trento

April 2014

Acknowledgements

Doing a Ph.D. is not an easy task as it might seem to someone. In my case, it was a taught and long process with lots of barriers. Without the support and advices of my relatives, friends and colleagues I would have never reached the endpoint of this journey. At the end of my thesis, it is a pleasant task to express my thanks to all those who contributed in many ways to the success of this study and made it an unforgettable experience for me.

First of all, I would like to thank my advisor Prof. Bruno Crispo for involving me into the life of a researcher, for his support, good advices and fair feedback. Without his supervision and constant help this dissertation would not have been possible.

During my PhD study I was lucky to work with other brilliant people: Mauro Conti, Giovanni Russello and Fabio Massacci. I would like to thank them for their advices and our discussions that helped to improve the quality of our works. Special thanks go to Earlence Fernandes for introducing me into the world of the Android system development. I offer my sincerest gratitude to Olga Gadyatskaya. Our collaboration has been being very fruitful and I hope that our friendship will last for years.

I gratefully acknowledge my close friends: Siarhei Bykau, Maksim Khadkevich, Heorhi Raik, and Ivan Tankoyeu. Their advices, support and encouragement have helped me to pass this long journey. Together with Liudmila Palavinka, Dina Shakirova, and Viktoria Tankoyeva they have become my Trento family. I would like to thank my friend Dmitry Krivonos for not letting me down during the hard time. He showed me that every problem could be solved, you just should not give up and keep searching for possibilities. I wish to thank my flatmates Anton Philippov, Evgeniy Borovin, and Alexandr Garaga. In addition, the words of gratitude go to the russian-speaking community in Trento and to all my friends who stay in Belarus. I remember you all!

I would like to thank my sister. Her restless care, deep trust and support in all my pursuits cannot be overestimated. Last but not least, my deep sincerest words of love go to my parents. Throughout all my study at the University and all my life, they cheer me on, celebrate each my accomplishment, and help me get through the hard periods of my life in the most positive way. My dear sister, my Mom, and Dad, I love you and wish you all the happiness!

Abstract

During the last few years mobile phones have been being replaced by new devices called smartphones. A more "intelligent" version of a mobile phones, smartphones combine usual "phoning" facilities with the functionality and performance of personal computers. Moreover, they are equipped with various sensors, such as camera and GPS, and are open to third-party applications.

Being almost all the time with their users, it is not surprising that smartphones have access to very sensitive private data. Unfortunately, these data are of particular interest not only to the device owners. Developers of third-party applications embed data collection functionality either to feed advertising frameworks or for their own purposes. Moreover, there are also adversaries aiming at gathering personal user information or performing malicious actions. In this situation the users have a strong motivation to safeguard their devices from being misused and want to protect their privacy.

Among all operating systems for mobile platforms, Android developed by Google is the recognized leader. This operating system is installed on four out of five new devices. In this thesis we propose a set of improvements to enhance security of the Android ecosystem and ensure trustworthiness of the applications installed on the device. In particular, we focus on the application ecosystem security, and research the following key aspects: identification of suspicious applications; application code analysis for malicious functionality; distribution of verified applications to end-user devices; and enforcing security on the device itself.

It was previously shown that adversaries often relied on app repackaging to burst the proliferation of malicious applications. As the first contribution, this dissertation proposes a fast approach to detect repackaged Android applications. If a repackaged application is detected, it is necessary to understand whether it is malicious or not. Today Android malware conceal their malicious nature using dynamic code update techniques, thus, static analyzers cannot detect this vicious behavior. The second contribution of this work is a static-dynamic analysis approach to discover and analyse apps in the presence of dynamic code updates routines. To increase the user's confidence in the installed apps, as the third contribution we propose the concept of trusted stores for Android. Our approach ensures that a user can install only the applications vetted and attested by trusted stores. Finally, the forth contribution is the design and implementation of a policy-based framework for enforcing software isolation of applications and data that may help to improve the security of end-user devices.

Keywords
Smartphones; Android; security; repackaging; malware; static-dynamic analysis

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 The Importance of Data Stored on Mobile Devices

Over the last decade the popularity of mobile phones has increased greatly. Today, on average, there is almost one phone per human being. Harnessing the rapid development of microelectronics, mobile phone manufacturers have started to produce more and more powerful equipment, leading to so called smartphones, which are now almost computationally equal to modern desktop computers.

Smartphone is a device which combines the functionality of a cell phone and a personal computer. Checking emails, browsing the Internet, taking pictures, making video and sharing the content are successfully carried out with the help of these gadgets. Furthermore, smartphones are full of different sensors that enrich the user experience. This situation resulted in very sensitive end-user data accumulated by smartphones. Unfortunately, these data are of particular interest not only to the device owners. Also developers of third-party applications collect data of their users [70, 84]. For example, advertising frameworks collect data to profile customers and provide targeted and customized ads. Moreover, there are also adversaries who develop applications to gather personal information and perform malicious actions [156]. Not surprisingly, in this situation the device owners have a strong interest in protecting their devices from being misused and want to control the dissemination of personal data.

Smartphones are also now used as business tools. Being extensively used by users in their daily life, not surprisingly they are also exploited in their daily business workflow. The market research [53] shows that more and more companies provide the employees with a possibility to use their own devices for business purposes. This trend is called *Bring Your Own Device* (BYOD). The benefits of using personal devices for business purposes are obvious for companies. This allows them to increase productivity and satisfaction of employees along with decreasing the expenditures for the infrastructure [53]. With the BYOD wave the problem how to integrate personal devices with the ability to perform

business tasks while ensuring security of business data also has also appeared.

Among all operating systems for smartphones we distinguish Android. This operating system was initially developed by Android Inc., which was later acquired by Google. Only 6 years after the appearance of the first Android-based device this platform is installed on about 82% of all new smartphones [20]. What is more important, this operating system is open-source, i.e., it is possible to develop new features, test them on real devices and submit for review, thus, contributing the improvements to the community. Moreover, this allows researchers to explore easily the platform for breaches in the architecture. Apparantly, lately this operating system has received a lot of attention in the security research community; and Android has rapidly become a reference mobile platform for research experiments.

These are the preliminary observations from which we begin the refinement of the problems investigated in this dissertation.

1.2 Problem Statement

Although Android has been developed with security in mind, this operating system itself, along with the ecosystem, still have some flaws and limitations that may contribute to personal data leakage. For example, the permission system[1], which is one of the main security mechanisms on Android, appears to be not very user-friendly. First of all, permissions are not fine-grained. Secondly, they are assigned to an application only during the installation based on the "all-or-nothing" principle, meaning that if a user does not agree with all requested permissions she will not be able to install the app. Thirdly, despite the importance of this security mechanism, recent studies (e.g., executed by Felt et al. [75]) show that users pay little attention to the validation routines provided by the platform: only 17% of users check application permissions during installation and only 3% understand permissions. Finally, permissions cannot be revoked at runtime. Along with the absence of so-called "prompt" permissions (which require the user to grant access to a senstivie feature at runtime), all these problems create a very fruitful environment for applications that want to steal user's data.

Given the high value of personal data stored on mobile device, it is clear that adversaries were also attracted. Yet, smartphone security mechanisms were proven to offer limited protection against applications that can leak data. This poses a serious threat to sensitive data, especially in case of corporate information. For example, malicious applications can access emails, SMS and MMS containing confidential data of the company and the end-user. Even more worrying is the number of *legitimate* applications harvesting the data that are not strictly necessary for the functions the applications advertise to users [70,84].

[1]The permission system is a security control to guard access to sensitive device features and data. More details will be provided later on.

Moreover, the current architecture of the Android operating system does not offer a possibility to enforce additional security policies, besides those offered by the platform by default. For example, using different sensors a smartphone could recognise the environment, and could use this context to restrict the access to some information or capabilities of a smartphone. This feature is especially important when a smartphone is used both as a personal and business device. The recent market research [53] shows that more and more companies allow the employees to use their own devices for business purposes. This trend is a win-win situation both for employees and employers. From the business owner point of view, this increases the productivity of the employees. From employees' perspective, the permission to use personal devices for executing business-related tasks allows to be more efficient and increases gratification. According to [53], 69% of IT leaders support the development of the BYOD trend. However, in this case the company wants to control the behavior of devices during the working hours, while the users want to have total control over them during the after office time.

All these facts lead us to the first issue raised in this thesis.

Issue 1 *How to provide several virtual environments controlled by different parties on the same real device? How to separate data and apps that belong to different sides of user's life? How to ensure the control over virtual environments for the owners of these environments?*

All these factors drive the research of Android security improvements on the operating system side. However, there are also some limitations in the Android ecosystem. Due to the openness of Android, third-party applications can be installed on a smartphone without any limitations. While other players, such as Apple and RIM, addressed the problem by running their certification schemes and vetting each application before being published on their own (unique) market, Google chose a different path. In the spirit of openness to third-party developers, Android apps do not need to be certified before being published on any market. It is user's responsibility to validate that the installed application will not harm the device and will not steal user private information. Unfortunately, ordinary users cannot make an unsupervised decision if an app can be trusted or not. However, they want to be reassured that the app has passed a vetting process and does not contain any harmful code that can steal personal data.

These aspects show a great demand for an app attestation service that will distribute only benign applications. Moreover, companies may also want to ensure that employees install only attested apps on their smartphones used in the business environment. Regrettably, due to the open nature of Android ecosystem, currently there are no facilities on Android to provide this service. Therefore, the second issue investigated in this thesis is:

Issue 2 *How to provide an attestation service for the Android platform while maintaining the open nature of the Android ecosystem?*

All this, in turn, requires the research and development of verification techniques that can be used to vet an app. Recently a lot of static analysis tools for Android applications have appeared, for instance, [71, 152], to mention a few. Unfortunately, the analysis by static tools can be easily evaded by exploiting dynamic code update features available for app developers in Android. For instance, the *Dynamic Class Loading* facility may be used to conceal virulent code in a separate file that is downloaded and executed runtime. Thus, during static analysis of such application this malicious code is unreachable, and, therefore, this app may be falsely marked as benign. Moreover, adversaries can postpone the download of harmful payload until an application is actually published on a market, thus, evading the dynamic analysis [117] performed by some markets (e.g., Google Play checks applications using dynamic analysis service called Bouncer [107]). Another technique, *Reflection*, is used to hide the name of a called function, i.e., the name of a function may be available for an application only during runtime. This technique may help an adversary to bypass static analysers that rely on *Method Call Graph* of an application. Not surprisingly, that the most advanced malware [99,155] use these techniques to conceal the malicious behavior. Unfortunately, dynamic code update techniques cannot be simply prohibited to use, because they are also widely exploited in benign apps. Thus, there is a strong need for a tool which may help to analyse the applications with dynamic code update features; and this is the third issue investigated in this thesis.

Issue 3 *How to analyse Android applications in the presence of dynamic code update features?*

However, to affect users with malicious applications the intruders need an efficient way to distribute their apps. Modern mobile devices are equipped with processors of different architectures, for instance, ARM, x86, MIPS. These processors operate on different instruction sets, so the executables of applications should be compiled separately for each of this platform. Android was designed to be used on a large variety of devices. It uses the Dalvik virtual machine to interpret on a particular architecture the bytecode produced during the build of an Android app. Thus, it is possible to produce one executable that may be run on different platforms. Unfortunately, this bytecode can be easily reverse engineered, changed and recompiled. This process in case of Android applications is called *repackaging*. Moreover, due to the peculiarities of Android ecosystem (presence of third-party markets and the usage of self-signed certificates to sign packages), de-facto there are no obstacles to perform repackaging of the vast majority of Android apps. Any developer may take an application, decompile it, change its parts or add her own code, recompile it and publish under her credentials in a market.

Not surprisingly, the repackaging is a very popular channel to distribute malicious apps. The study of Android malware [156] shows that about 86% of malware samples are the repackaged versions of legitimate apps. Additionally to the distribution of malware, repackaging may be used to steal money from the original developer. Recent analysis [85] shows that the developers of original apps loose about 14% of advertisement revenues and about 10% of the user base due to application plagiarism. All these facts show that the Android ecosystem suffers from the intruders who repackage benign applications and distribute them through third-party markets. Hence, the detection of repackaged applications and their removal at least from the official market is a first-order task. Regrettely, there are more than 1 million apps [139] only in the official Google Play [22] market. Along with the apps distributed over numerous third-party markets, this creates an incredibly huge dataset. In addition, each application may have several versions. All these factors require an approach that can detect potentially repackaged applications in a fast way. This is the fourth issue investigated in this thesis.

Issue 4 *How to detect repackaged Android applications in a fast and efficient way?*

1.3 Thesis Contributions

This work focuses on different aspects how to improve security of the Android operating system and its ecosystem. The solutions proposed in this thesis are tightly interwoven with the modifications of the Android OS. The issues raised in Section 1.2 are covered in the reversed order. This approach allows us to comply with the usual workflow in security, when the threats are identified, then analyzed, and after this mitigated. Similarly, in this work we at first identify potential malicious applications (repackaged apps), then analyze them for dynamic code update features, and later propose the solutions how to mitigate the threat of malicious applications both on the market and on the device levels. The thesis is structured in the way to highlight the main contributions done during the Ph.D. study.

1.3.1 Fast Detection of Repackaged Android Applications

To address Issue 4 we developed a system called FSquaDRA. The system uses an approach to detect repackaged applications based on comparison of the resource files constituting an Android app package. The approach is based on the observation that malicious repackagers usually do not change the resources of an Android package because they want to resemble the original app as much as possible. Therefore, the code files may change during the repackaging, while the resource files (icons, drawables, music and video files, etc.) usually remain the same. Using the peculiarities of Android app signing process we

designed and implemented a very fast algorithm that can be used for pairwise comparison of apps.

This contribution will appear in the paper [149]:

- Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina and E. Moser. "FSquaDRA: Fast Detection of Repackaged Applications". In *Proceedings of the 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec'14)*, to appear in 2014.

1.3.2 Static-Dynamic Analyser of Android Apps in the Presence of Reflection and Dynamic Class Loading

When suspicious candidates are identified, for instance, using the approach described in Section 1.3.1, they have to be scrutinized with analysis tools. Unfortunately, static analysis of Android applications can be hindered by the presence of the popular dynamic code update techniques inherited from Java: dynamic class loading and reflection. For example, recent Android malware samples specifically use dynamic code update routines to conceal their malicious behavior from static analysis. These techniques defuse even the most recent static analyzers (e.g., [68, 152]), which explicitly make the closed world assumption.

To address Issue 3 we propose an approach that augments the information available to static analyzers. It combines static and dynamic analysis of Android applications in order to reveal the hidden/updated behavior, and extends the method call graph of the application under analysis with this information. Moreover, our approach allows to unroll the suspicious behavior located in the nested code files.

1.3.3 Attestation Service for the Android Platform

In the Android ecosystem the process of verifying the integrity of downloaded apps is left to the user. Different from other systems, e.g., Apple App Store, Google does not provide any certified vetting process for the Android apps. To address Issue 2 we designed an architecture called TruStore that enables the deployment of application certification service on the Android platform. In our approach, the TruStore client enabled on the end-user device ensures that only the applications, which have been certified by the TruStore server, are installed on the user smartphone. We envisage trusted markets (TruStore servers, which can be, e.g., corporate application markets) that guarantee security by enabling an application vetting process. For instance, applications can be vetted using our systems FSquaDRA and StaDynAdescribed in this thesis. The TruStore infrastructure maintains the open nature of the Android ecosystem and requires minor modifications to the Android stack. Moreover, it is backward-compatible and transparent

for developers, and does not change the application management process on a device. Thus, the proposed approach allows us to include the vetting of Android applications into the workflow, and provides a possibility to ensure that only vetted apps are installed on user devices.

The conducted research is released as the demo-paper [147],while more information can be found in the technical report [148]:

- Y. Zhauniarovich, O. Gadyatskaya and B. Crispo. "DEMO: Enabling Trusted Stores for Android". In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security (CCS)*.

1.3.4 Supporting and Enforcing Security Profiles in Android

To address Issue 1 we implemented MOSES, a policy-based framework for enforcing software isolation of applications and data on the Android platform. In MOSES, it is possible to define distinct *Security Profiles* within a single smartphone. Each *Security Profile* is associated with a set of policies that control the access to applications and data. Moreover, each *Security Profile* can have an associated trusted third-party that specifies the policies for the profile. Thus, a user may use MOSES for protecting the device against malicious applications specifying the security policy for her personal profile, while the policy for the device usage at work may be defined by the IT experts of the company. As profiles are not predefined or hardcoded, they can be specified and applied at any time. One of the main features of MOSES is the dynamic switching from one *Security Profile* to another. This allows MOSES to automatically start enforcing the rules defined in the policies for the profiles. This functionality, for instance, may be used by business owners who want to ensure that entertainment and social network applications are not used by their employees during the working hours.

This contribution is discussed in details in the paper [150], while the demo of the system is presented in [124]:

- Y. Zhauniarovich, G. Russello, M. Conti, B. Crispo and E. Fernandes. "MOSES: Supporting and Enforcing Security Profiles on Smartphones". In *IEEE Transactions on Dependable and Secure Computing*, to appear in 2014.

- G. Russello, M. Conti, B. Crispo, E. Fernandes and Y. Zhauniarovich. "Demonstrating the Effectiveness of MOSES for Separation of Execution Modes". In *Proceedings of the 2012 ACM SIGSAC conference on Computer & Communications Security (CCS)*.

1.4 Accepted Papers

1. [149] – Y. Zhauniarovich, O. Gadyatskaya, B. Crispo, F. La Spina and E. Moser. "FSquaDRA: Fast Detection of Repackaged Applications". In *Proceedings of the 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy (DBSec'14)*, to appear in 2014.
2. [150] – Y. Zhauniarovich, G. Russello, M. Conti, B. Crispo and E. Fernandes. "MOSES: Supporting and Enforcing Security Profiles on Smartphones". In *IEEE Transactions on Dependable and Secure Computing*, to appear in 2014.
3. [79] – O. Gadyatskaya, F. Massacci and Y. Zhauniarovich. "Security in the Firefox OS and Tizen Mobile Platforms". In *IEEE Computer*, to appear in 2014.
4. [147] – Y. Zhauniarovich, O. Gadyatskaya, and B. Crispo. "DEMO: Enabling Trusted Stores for Android". In *Proceedings of the 2013 ACM SIGSAC conference on Computer & Communications Security (CCS)*.
5. [124] – G. Russello, M. Conti, B. Crispo, E. Fernandes and Y. Zhauniarovich. "Demonstrating the Effectiveness of MOSES for Separation of Execution Modes". In *Proceedings of the 2012 ACM SIGSAC conference on Computer & Communications Security (CCS)*.
6. [62] – M. Conti, B. Crispo, E. Fernandes and Y. Zhauniarovich. "CRêPE: A System for Enforcing Fine-Grained Context-Related Policies on Android". In *IEEE Transactions on Information Forensics and Security*, 2012.
7. [125] – G. Russello, B. Crispo, E. Fernandes and Y. Zhauniarovich. "YAASE: Yet Another Android Security Extension". In *Proceedings of the 2011 IEEE Third International Conference on Privacy, Security, Risk and Trust (PASSAT) and the 2011 IEEE Third International Conference on Social Computing (SocialCom)*, 2011.

1.5 Thesis Structure

The thesis is structured to present the main contributions of the Ph.D. study. The rest of the dissertation is organized as follows:

Chapter 2 discusses the security background of the Android operating system. The deep understanding of the security mechanisms implemented in this operating system sets the context for the reader to better understand the Android limitations and the approaches solving them proposed in this work.

Chapter 3 presents FSquaDRA, an approach relying on the peculiarities of the Android app signing process that we propose for fast detection of repackaged applications.

Chapter 4 observes the problem of dynamic code updates in Android applications and proposes a tool that, using a combination of static and dynamic analysis techniques, is able to analyze Android applications in the presence of dynamic class loading and reflection routines.

Chapter 5 researches how an attestation service can be implemented on the Android platform. In this chapter we discuss how to ensure trustworthiness of Android apps providing additional evidences that an application has been vetted.

Chapter 6 indicates how *Security Profiles* can be implemented on Android. It also presents a framework that can be used to define fine-grained security policies for each *Security Profile* enabling protection against applications that harvest sensitive information. Additionally, the task of security policy enforcement depending on the context is also considered in this chapter.

Chapter 7 recaps the main contributions described in this dissertation and reveals the future work.

Chapter 2

Android Security

The comprehension of the Android security architecture is an essential premise for the understanding the issues and the solutions covered in this thesis. This section considers the basics of the Android architecture from the security perspective.

Code examples in this chapter are provided for Android 4.2.2_r1.2 version and for Androlized Linux kernel 3.4 version.

2.1 Android Stack

Android is a software stack for a wide range of mobile devices and a corresponding open-source project led by Google [38]. Android consists of four layers: *Linux Kernel*, *Native Userspace*, *Application Framework* and *Applications*. Sometimes *Native Userspace* and *Application Framework* layers are combined into the one called *Android Middleware*. Figure 2.1 represents the layers of the Android software stack. Roughly saying, in this figure the green blocks correspond to the components developed in C/C++, while the blue cohere with the ones implemented in Java. Google distributes the most part of the Android code under Apache version 2.0 licence. The most notable exception to this rule is the changes in the *Linux Kernel*, which are under GNU GPL version 2 licence.

Linux Kernel. Before being acquainted by Google in 2005, Android was a startup product of the Android Inc. company. One of the features of startup companies is their tendency to maximise the reuse of already existing components to reduce the time and the cost of their product. So did Android Inc. selecting the *Linux Kernel* as a centerpiece of their new platform. In Android, *Linux Kernel* is responsible for process, memory, communication, filesystem management, etc. While Android mostly relies on the "vanilla" Linux Kernel functionality, several custom changes, which are required for the system operation, have been proposed to this level. Among them *Binder* (a driver, which provides the support for custom RPC/IPC mechanism in Android), *Ashmem* (a replacement of the standard

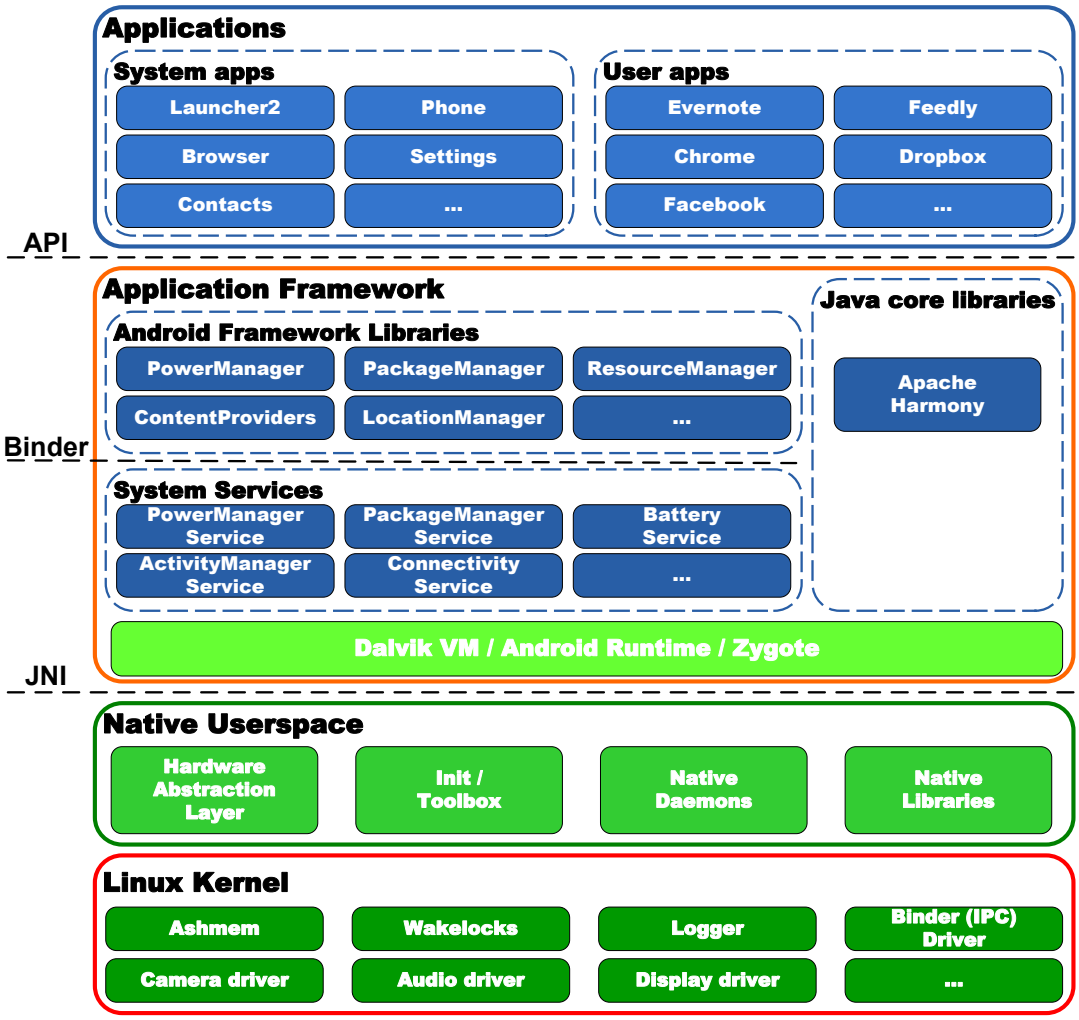


Figure 2.1: Android software stack

Linux shared memory functionality), *Wakelocks* (a mechanism that prevents the system from going to sleep) are the most notable ones [144]. Although these changes proved to be very useful in case of mobile operating systems, they are still out of the main branch of *Linux Kernel*.

Native Userspace. By the *Native Userspace* we understand all userspace components that run outside *Dalvik Virtual Machine* and do not belong to the *Linux Kernel* layer. The first component of this layer is *Hardware Abstraction Layer (HAL)* that is actually blurred between the *Linux Kernel* and *Native Userspace* layers. In Linux, drivers for hardware are either embedded into the kernel or loaded dynamically as modules. Although Android is built on top of *Linux Kernel* it exploits a very different approach to support new hardware. Instead, for each type of hardware Android defines an API that is used by upper layers

to interact with this type of hardware. The suppliers of a hardware must provide a software module that is responsible for the implementation of the API defined in Android for this particular type of hardware. Thus, this solution allows Android not to embed all possible drivers into the kernel anymore and to disable the dynamic module loading kernel mechanism. The component, which provides this functionality, has been called *Hardware Abstraction Layer* in Android. Additionally, such architectural solution lets hardware suppliers to select the licence, under which their drivers are distributed [143, 144].

Kernel finishes its booting by starting only one userspace process called *init*. This process is responsible for starting all other processes and services in Android, along with performing some operations in the operating system. For instance, if a critical service stops answering in Android, the *init* process can reboot it. This process performs operations in accordance to the `init.rc` configuration file. *Toolbox* includes essential binaries, which provide shell utilities functionality in Android [144].

Android also relies on a number of key daemons. It starts them during system startup and preserves them running, when the system is working. For instance, *rild* (the Radio Interface Layer daemon, responsible for communications between baseband processor and other system), *servicemanager* (a daemon, which contains an index of all Binder services running in Android), *adbd* (Android Debug Bridge daemon that serves as a connection manager between host and target equipment), etc.

The last but not least component in *Native Userspace* is *Native Libraries*. There are two types of *Native Libraries*: native libraries that come from external projects, and developed within Android itself. These libraries are loaded dynamically and provide various functionality for Android processes [144].

Application Framework. *Dalvik* is Android's registry-based virtual machine. It allows the operating system to execute Android applications, which are written using Java language. During the built process, Java classes are compiled into a `.dex` file that are interpreted by the *Dalvik VM*. The *Dalvik VM* was specifically designed to be run in constrained environments. Additionally, the *Dalvik VM* provides functionality to interact with the rest of the system, including native binaries and libraries. To accelerate the process initialization procedure Android exploits a specific component called *Zygote*. This is a special "pre-warmed" process that has all core libraries linked in. When a new app is about to run, Android forks a new process from *Zygote* and sets the parameters of the process according to the specification of the launched application. This solution allows the operating system not to copy linked libraries into a new process, thus, speeding up app launching operation. *Java Core Libraries*, which are used in Android, are borrowed from Apache Harmony project.

System Services is one of the most important parts of Android. Android comes with a number of *System Services* that provide basic mobile operating system func-

tionality to be used by Android app developers in their applications. For instance, `PackageManagerService` is responsible for managing (installation, update, deletion, etc.) Android packages within the operating system. Using *JNI* interfaces system services can interact with the daemons, toolbox binaries and native libraries of the *Native Userspace* layer. The public API to *System Services* is provided via *Android Framework Libraries*. This API is used by application developers to interact with *System Services*.

Android Applications. Android applications are software applications that run on Android and provide most of the functionality available for the user. The stock Android operating system is shipped with a number of built-in apps called *System Applications*. These are applications compiled as a part of AOSP built process. Moreover, the user may install *User Applications* from numerous app markets to extend basic and introduce new functionality to the operating system.

2.2 Android General Security Description

The core security principle of Android is that an adversary app should not harm the operating system resources, the user and other applications. To procure the execution of this principle, Android being a layered operating system, exploits the provided security mechanisms of all the levels. Focusing on security, Android combines two levels of enforcement [72, 129]: at the *Linux Kernel* level and at the *Application Framework* level (see Figure 2.2).

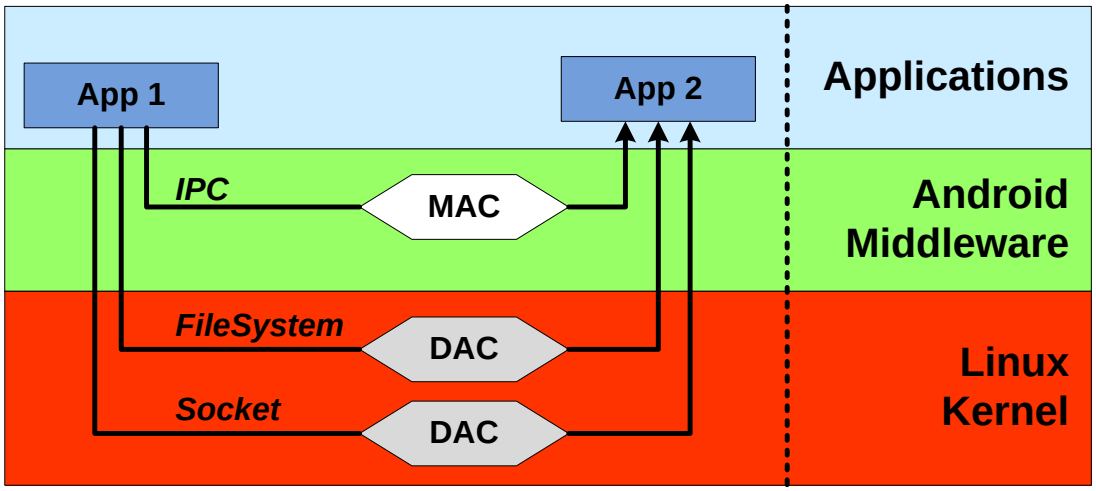


Figure 2.2: Two levels of Android security enforcement

At the *Linux Kernel* level each application is run in special *Application Sandbox*. The kernel enforces the isolation of applications and operating system components exploiting standard Linux facilities (process separation and Discretionary Access Control over

network sockets and filesystem). This isolation is imposed by assigning each application a separate Unix user (UID) and group (GID) identifiers. Such architectural decision enforces running each application in a separate Linux process. Thus, due to *Process Isolation* implemented in Linux, by default applications cannot interfere each other and have limited access to the facilities provided by the operating system. Therefore, *Application Sandbox* ensures that an application cannot drain the operating system resources and cannot interact with other apps [6].

The enforcement mechanism provided at the *Linux Kernel* layer effectively sandboxes an application from other apps and the system components. At the same time, an effective communicating protocol is required to allow developers to reuse application components and interact with the operating system units. This protocol is called *Inter-Process Communication* (IPC) because it facilitates the interactions between different processes. In case of Android, this protocol is implemented at the *Android Middleware* level (with a special driver released on the *Linux Kernel* level). The security on this level is provided by the *IPC Reference Monitor*. The reference monitor mediates all communications between processes and controls how applications access the components of the system and other apps. In Android, *IPC Reference Monitor* follows *Mandatory Access Control* (MAC) access control type.

All Android apps by default are run in low-privileged *Application Sandboxes*. Thus, an application has an access only to a limited set of system capabilities. The Android operating system controls the access of apps to the system resources that may adversely impact user experience [6]. This control is implemented in different forms, some of them are considered in details in Sections 2.3, 2.4 and 2.5. There is also a subset of protected system features (for instance, camera, telephony or GPS functionality), the access to which should be provided to third-party apps. However, this access should be provided in controlled manner. In case of Android, such control is realized using *Permissions*. Basically, each sensitive API, which provides access to the protected system resources, is assigned with a *Permission* – unique security label. Moreover, protected features may also include components of other applications.

To make the use of protected features, the developer of an application must request the corresponding permissions in the special `AndroidManifest.xml` manifest file. During the installation of an application the Android OS parses this file and presents the user with the list of the permissions declared in this file. The installation of an application occurs according to "all or nothing" principle, meaning that the app is installed only if all permissions are accepted. Otherwise, the application will not be installed. The permissions are granted only at the installation time and can not be modified later. As an example of a permission, consider an application that needs to monitor incoming SMS messages. In this case, the `AndroidManifest.xml` file must contain in the `<uses-permission>` tag the following declaration: `"android.permission.RECEIVE_SMS"/>`.

An attempt of an application to use a feature, which permission has not been declared in the *Android Manifest* file, will typically result in a thrown security exception. The details of permission enforcement mechanism we consider in the following sections.

More detailed security architecture of Android is shown in Figure 2.3. We will refer to it here and there in this work to explain the peculiarities of this operating system.

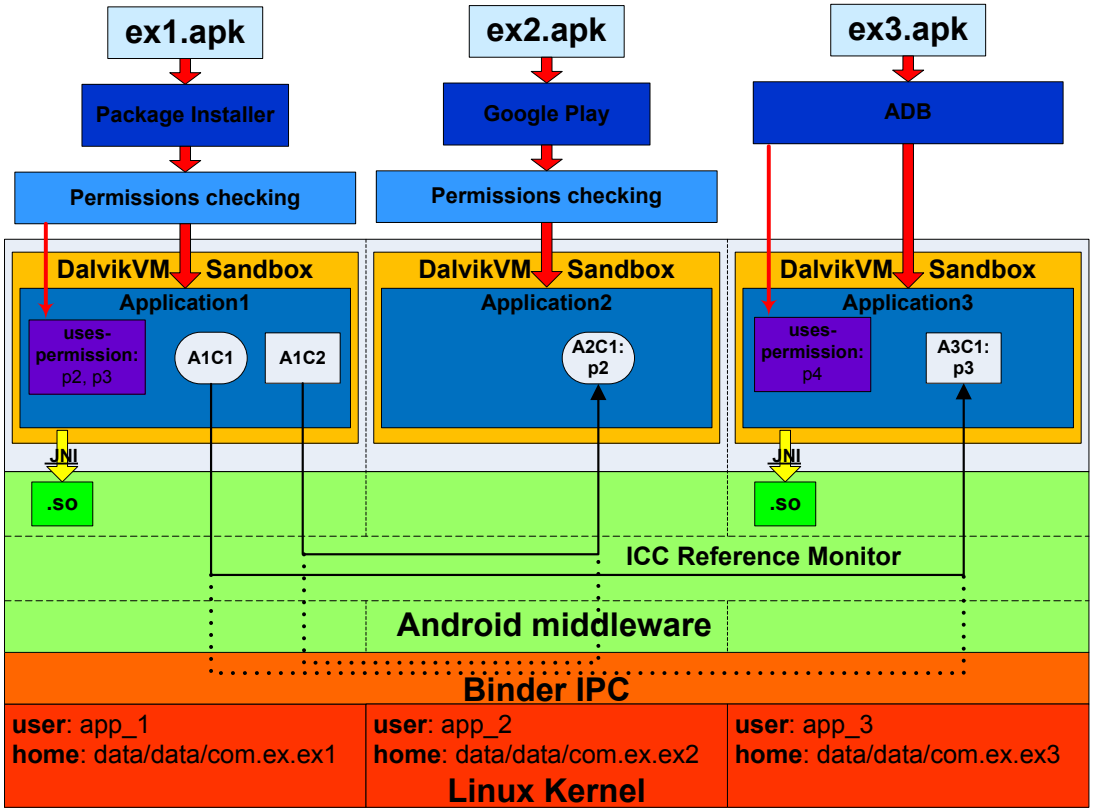


Figure 2.3: Android security architecture

2.3 Android Security on the Linux Kernel Level

One of the most widely known open-sources projects, Linux has proved itself as a secure, trusted and stable piece of software being researched, attacked and patched by thousands of people all over the world. Not surprisingly, *Linux Kernel* is the basis of the Android operating system [6]. Android relies on Linux not only for process, memory and filesystem management, it is also one of the most important components in the Android security architecture. In Android *Linux Kernel* is responsible for provisioning *Application Sandboxing* and enforcement of some permission.

2.3.1 Application Sandboxing

Let consider the process of an Android application installation in details. Android apps are distributed in the form of *Android Package* (`.apk`) files. A package consists of a Dalvik executable, resources, native libraries and a manifest file, and is signed by a developer signature. There are three main mediators that may install a package on a device in the stock Android operating system:

- **Google Play**.
- **Package Installer**.
- **adb install**.

Google Play is a special application that provides the user with a capability to look for an application uploaded to the market by third-party developers along with a possibility to install it. Although it is also a third-party application, Google Play app (because of being signed with the same signature as the operating system) has access to protected components of Android, which other third-party applications lack for. In case if the user installs applications from other sources she usually implicitly uses *Package Installer* app. This system application provides an interface that is used to start package installation process. The utility *adb install*, which is provided by Android, is mainly used by third-party app developers. While the former two mediators require a user to agree with the list of permissions during the installation process, the latter installs an app quietly. That is why it is mainly used in developer tools aiming at installing an application on a device for testing. This process is shown in the upper part of Figure 2.3.

The process of provisioning *Application Sandbox* at the Linux kernel level is the following. During the installation, each package is assigned with a unique user identifier (*UID*) and a group identifier (*GID*) that are not changed during app life on a device. Thus, in Android each application has a corresponding Linux user. User name follows the format `app_x`, and *UID* of that user is equal to `Process.FIRST_APPLICATION_UID + x`, where `Process.FIRST_APPLICATION_UID` constant corresponds to 10000. For instance, in Figure 2.3 `ex1.apk` package during the installation receives `app_1` user name, and *UID* equal to 10001 .

In Linux, all files in memory are subject for Linux *Discretionary Access Control* (DAC). Access permissions are set by a creator or an owner of a file for three types of users: the owner of the file, the users who are in the same group with the owner and all other users. For each type of users, a tuple of read, write and execute (r-w-x) permissions are assigned. Hence, so as each application has its own UID and GID, Linux kernel enforces the app execution within its own isolated address space. Beside that, the app unique UIDs and GIDs are used by Linux kernel to enforce fair separation of device resources (memory, CPU,

etc.) between different applications. Each application during the installation also receives its own home directory, for instance, `/data/data/package_name`, where `package_name` is the name of an Android package, for example, `com.ex.ex1` In terms of Android, this folder is *Internal Storage*, where an application keeps its private data. Linux permissions assigned to this directory allows only the "owner" application to write to and read from this directory. It should be mentioned that there are some exceptions. The apps, which are signed with the same certificate, are able to share data between each other, may have the same UID or can even run in the same process.

These architectural decisions set up effective and efficient *Application Sandbox* on the *Linux Kernel* level. This type of sandbox is simple and based on the verified Linux *Discretionary Access Control* model. Luckily, so as the sandbox is enforced on the *Linux Kernel* level, native code and operating system applications are also subject to these constraints described in this section [6].

2.3.2 Permission Enforcement on the Linux Kernel level

It is possible to restrict the access to some system capabilities by assigning the Linux user and group owners to the components that implement this functionality. This type of restrictions can be applied to system resources like files, drivers and sockets. Android uses *Filesystem Permissions* and Android-specific kernel patches (known as *Paranoid Networking*) [81] to restrict the access to low-level system features like network sockets, camera device, external storage, possibility to read logs, etc.

Using filesystem permissions to files and device drivers, it is possible to limit processes in accessing some functionality of a device. For instance, such technique is applied to restrict access of applications to a device camera. The permissions to `/dev/cam` device driver is set to `0660`, with `root` owner and `camera` owner group. This means that only processes run as `root` or which are included in `camera` group, are able to read from and write to this device driver. Thus, only applications, which are included into `camera` group can interact with the camera. The mappings between permission labels and corresponding groups are defined in the file `frameworks/base/data/etc/platform.xml`, which excerpt is presented in Listing 2.1. Thus, during the installation if an app has requested the access to a camera feature and the user has approved it, this application is also assigned a `camera` Linux group GID (see corresponding Lines 8 and 9 in Listing 2.1). Therefore, this app receives a possibility to read information from `/dev/cam` device driver.

There are several points in Android where filesystem permissions to files, drivers and unix-sockets are set in: *init* program, `init.rc` configuration file(s), `ueventd.rc` configuration file(s) and *system ROM* filesystem config file. They are considered in details in Section 2.4.

In traditional Linux distributions, all processes are allowed to initiate network connec-

```
1  ...
2  <permissions>
3  ...
4      <permission name="android.permission.INTERNET" >
5          <group gid="inet" />
6      </permission>
7
8      <permission name="android.permission.CAMERA" >
9          <group gid="camera" />
10     </permission>
11
12     <permission name="android.permission.READ_LOGS" >
13         <group gid="log" />
14     </permission>
15 ...
16 </permissions>
```

Listing 2.1: The mappings between permission labels and Linux groups

tions. At the same time, for mobile operating systems the access to networking capabilities has to be controlled. To implement this control in Android, special kernel patches have been added that limit the access to network facilities only to the processes that belong to specific Linux groups or have specific Linux capabilities. These Android-specific patches of the Linux kernel have obtained the name *Paranoid networking*. For instance, for `AF_INET` socket address family, which is responsible for network communication, this check is performed in `kernel/net/ipv4/af_inet.c` file (see the code extraction in Listing 2.2). The mappings between the Linux groups and permission labels for *Paranoid networking* are also set in `platform.xml` file (for instance, see Line 4 in Listing 2.1).

Similar *Paranoid Networking* patches are also applied to restrict the access to IPv6 and Bluetooth [144].

The constants used in these checks are hardcoded in the kernel and specified in the `kernel/include/linux/android_aid.h` file (see Listing 2.3).

Thus, at the *Linux Kernel* level the Android permissions are enforced by checking if an application is included into a special predefined group. Only the members of this group have access to the protected functionality. During the installation of an app, if a user has agreed with the requested permission, the application is included into the corresponding Linux group and, hence, receives access to the protected functionality.

2.4 Android Security on the Native Userspace Level

The *Native Userspace* level plays an important role in the security provisioning of the Android operating system. It is impossible to understand how the security architectural decisions are enforced in the system without the comprehension what happens on this layer. In this section the topics of the Android booting process and the filesystem peculiarities are considered along with the summarization how the security is enforced on the

```
...
#ifdef CONFIG_ANDROID_PARANOID_NETWORK
#include <linux/android_aid.h>

static inline int current_has_network(void)
{
        return in_egroup_p(AID_INET) || capable(CAP_NET_RAW);
}
#else
static inline int current_has_network(void)
{
        return 1;
}
#endif
...

/*
 *      Create an inet socket.
 */

static int inet_create(struct net *net, struct socket *sock, int protocol,
                       int kern)
{
    ...
    if (!current_has_network())
                return -EACCES;
    ...
}
```

Listing 2.2: Paranoid networking patch

Native Userspace level.

2.4.1 Android Booting Process

To understand what procedures provision security on the *Native Userspace* level, at first the booting sequence of an Android device should be considered. It should be mentioned that during the first steps this sequence may vary on different devices but after the Linux kernel is loaded the process is usually the same. The flow of the booting process is shown in Figure 2.4.

When a user powers on a smartphone the CPU of the device will appear in a non-initialised state. In this case, a processor starts executing commands beginning from a hardwired address. This address points to a piece of code in the write-protected memory of the CPU, where *Boot ROM* is located (see Step 1 in Figure 2.4). The main aim of the code resided on Boot ROM is to detect a media, where *Boot Loader* is located [134]. When the detection is done, Boot ROM loads Boot Loader into the internal memory (which is only available after device power-on) and performs a jump to the loaded code of Boot Loader. On its turn, *Boot Loader* sets up external RAM, filesystem and network support. After that it loads *Linux Kernel* into the memory and passes the execution to it. *Linux Kernel* initialises the environment to run C code, activates interrupt controllers, sets up memory management units, defines scheduling, loads drivers and mounts root

```
...
#ifndef _LINUX_ANDROID_AID_H
#define _LINUX_ANDROID_AID_H

/* AIDs that the kernel treats differently */
#define AID_OBSOLETE_000 3001  /* was NET_BT_ADMIN */
#define AID_OBSOLETE_001 3002  /* was NET_BT */
#define AID_INET         3003
#define AID_NET_RAW      3004
#define AID_NET_ADMIN    3005
#define AID_NET_BW_STATS 3006  /* read bandwidth statistics */
#define AID_NET_BW_ACCT  3007  /* change bandwidth statistics accounting */

#endif
```

Listing 2.3: Android id constants hardcoded in Linux kernel

filesystem. When memory management units are initialized, the system is ready to use virtual memory and run user-space processes [134]. Actually, starting from this step the process does not differ from the one that occurs on desktop computers running Linux.

The first user-space process, which is an ancestor of all processes in Android, is *init*. The executable of this program is located in the `root` directory of the Android filesystem. Listing 2.4 contains the focal points of the sources of this executable. It can be seen that the `init` binary is responsible for the creation of the basic filesystem entries (Lines from 7 to 16). After that (Line 18), the program parses the `init.rc` configuration file and executes the commands written there.

```
int main(int argc, char **argv)
{
...
    if (!strcmp(basename(argv[0]), "ueventd"))
        return ueventd_main(argc, argv);
...
    mkdir("/dev", 0755);
    mkdir("/proc", 0755);
    mkdir("/sys", 0755);

    mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755");
    mkdir("/dev/pts", 0755);
    mkdir("/dev/socket", 0755);
    mount("devpts", "/dev/pts", "devpts", 0, NULL);
    mount("proc", "/proc", "proc", 0, NULL);
    mount("sysfs", "/sys", "sysfs", 0, NULL);
...
    init_parse_config_file("/init.rc");
...
}
```

Listing 2.4: The sources of *init* program

The `init.rc` configuration file is written using a language called *Android Init Language* and located in the `root` directory. This configuration file can be imagined as a list of actions (sequence of commands), which execution is triggered by the predefined events. For instance, in Listing 2.5, `fs` (Line 1) is a *trigger*, while Lines 4 – 7 represent the *Actions*.

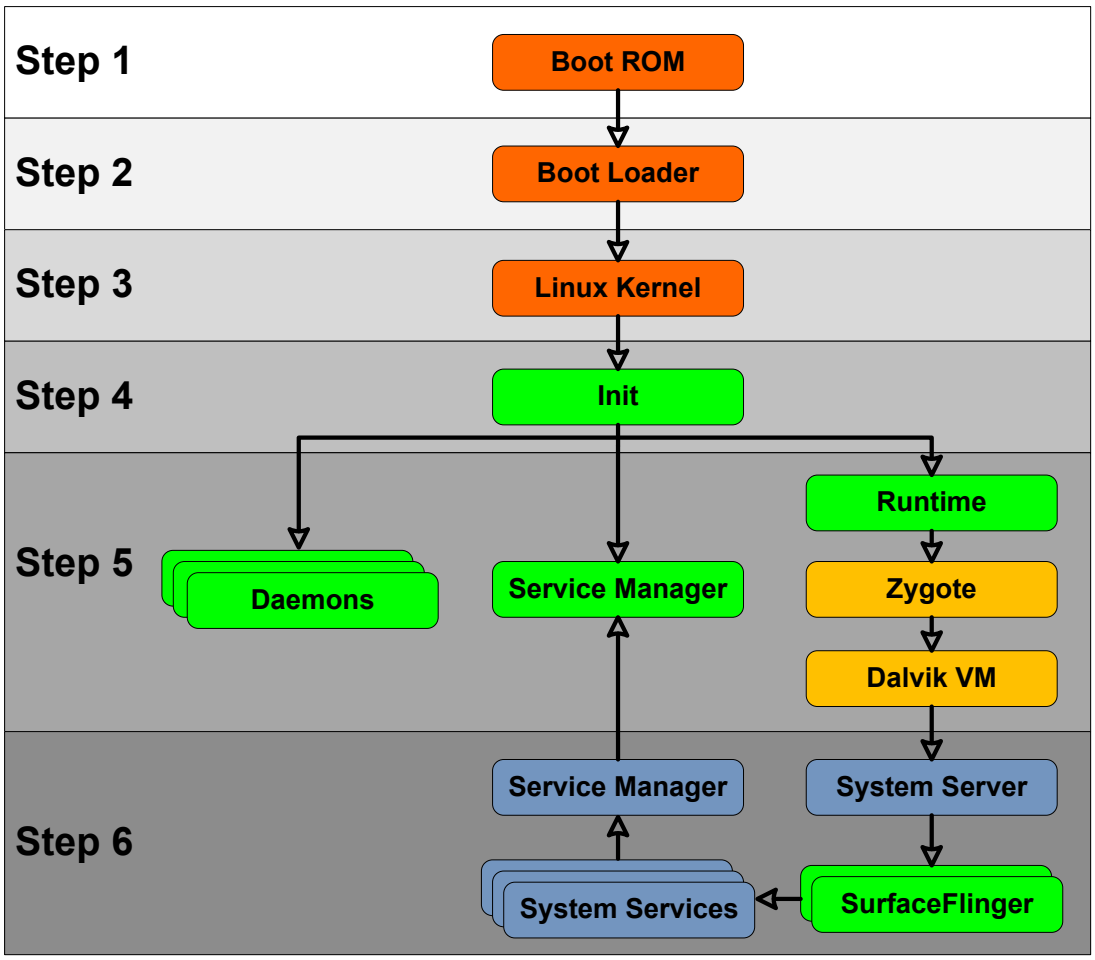


Figure 2.4: Android boot sequence

The commands written in the `init` configuration file defines system global variables, sets up basic kernel parameters for memory management, configures filesystem, etc. What is more important from the security perspective, it is also responsible for the basic filesystem structure creation and for the assignment of the owners and the filesystem permissions to the created nodes.

Additionally, the *init* program is responsible for starting several essential daemons and processes in Android (see Step 5 in Figure 2.4), the parameters of which are also defined in the `init.rc` file. An executed process in Linux by default is run with the same permissions (under the same UID) as an ancestor. In Android, *init* is started with the *root* privileges (`UID == 0`). This means that all descendant processes should run with the same UID. Luckily, the privileged processes may change their UIDs to the less privileged ones. Thus, all descendants of the *init* process may use this functionality specifying the UID and the GID of a forked process (the owner and group are also defined in the `init.rc` file).

```
1 on fs
2     # mount mtd partitions
3     # Mount /system rw first to give the filesystem a chance to save a checkpoint
4     mount yaffs2 mtd@system /system
5     mount yaffs2 mtd@system /system ro remount
6     mount yaffs2 mtd@userdata /data nosuid nodev
7     mount yaffs2 mtd@cache /cache nosuid nodev
```

Listing 2.5: The list of actions performed on `fs` trigger in emulator

One of the first daemons, which is forked from the *init* process, is the *ueventd* daemon. This service runs its own `main` function (see Line 5 in Listing 2.4) that reads the `ueventd.rc` and `ueventd.[device_name].rc` configuration files and replays the specified there kernel *uevent* hotplug events. These events set up the owners and permissions for different devices (see Listing 2.6). For instance, Line 5 shows how the filesystem permissions to `/dev/cam` device are set, which example was considered in Section 2.3. After that, the daemon waits listening for all future hotplug events.

```
1 ...
2 /dev/ashmem               0666   root       root
3 /dev/binder               0666   root       root
4 ...
5 /dev/cam                  0660   root       camera
6 ...
```

Listing 2.6: `ueventd.rc` file

One of the core services started by the *init* program is *servicemanager* (see Step 5 in Figure 2.4). This service acts as an index of all services running in Android. It must be available on early phase because all system services, which are started afterward, should have a possibility to register themselves and, thus, become visible to the rest of the operating system [144].

Another core process launched by the *init* process is *Zygote*. *Zygote* is a special process that has been warmed-up. This means that the process has been initialised and linked against the core libraries. *Zygote* is an ancestor for all processes. When a new application is started, *Zygote* forks itself. After that, the parameters corresponding to a new application, for instance, UID, GIDs, nice-name, etc., are set for the forked child process. The acceleration of a new process creation is achieved because there is no need to copy core libraries into the new process. The memory of a new process has "copy-on-write" protection, meaning that the data will be copied from the zygote process to a new one only if the latter tries to write into the protected memory. So as core libraries cannot be changed, they are remained only in one place reducing memory consumption and the application startup time.

The first process, which is run using *Zygote* is *System Server* (Step 6 in Figure 2.4). This process, at first, runs native services, such as *SurfaceFlinger* and *SensorService*.

After the services initialized, a callback is invoked, which starts the remaining services. All these services are then registered with *servicemanager*.

2.4.2 Android Filesystem

Although Android is based on *Linux Kernel*, its filesystem hierarchy does not comply with Filesystem Hierarchy Standard [41] that defines filesystem layout of Unix-like systems (see Listing 2.7). Some directories in Android and in Linux are the same, for instance, `/dev`, `/proc`, `/sys`, `/etc`, `/mnt`, etc. The purposes of these folders are the same as in Linux. At the same time, there are directories, such as `/system`, `/data` and `/cache`, which cannot be found in the Linux systems. These folders are the core parts of Android. During the build of the Android operating system, three image files are created: `system.img`, `userdata.img` and `cache.img`. These images provide the core functionality of Android and are the ones that are flashed on a device. During the boot of the system the *init* program mounts these images to the predefined mounting points, like `/system`, `/data` and `/cache` correspondingly (see Listing 2.5).

```
1  drwxr-xr-x root     root              2013-04-10 08:13 acct
2  drwxrwx--- system   cache             2013-04-10 08:13 cache
3  dr-x------ root     root              2013-04-10 08:13 config
4  lrwxrwxrwx root     root              2013-04-10 08:13 d -> /sys/kernel/debug
5  drwxrwx--x system   system            2013-04-10 08:14 data
6  -rw-r--r-- root     root          116 1970-01-01 00:00 default.prop
7  drwxr-xr-x root     root              2013-04-10 08:13 dev
8  lrwxrwxrwx root     root              2013-04-10 08:13 etc -> /system/etc
9  -rwxr-x--- root     root       244536 1970-01-01 00:00 init
10 -rwxr-x--- root     root         2487 1970-01-01 00:00 init.goldfish.rc
11 -rwxr-x--- root     root        18247 1970-01-01 00:00 init.rc
12 -rwxr-x--- root     root         1795 1970-01-01 00:00 init.trace.rc
13 -rwxr-x--- root     root         3915 1970-01-01 00:00 init.usb.rc
14 drwxrwxr-x root     system            2013-04-10 08:13 mnt
15 dr-xr-xr-x root     root              2013-04-10 08:13 proc
16 drwx------ root     root              2012-11-15 05:31 root
17 drwxr-x--- root     root              1970-01-01 00:00 sbin
18 lrwxrwxrwx root     root              2013-04-10 08:13 sdcard -> /mnt/sdcard
19 d---r-x--- root     sdcard_r          2013-04-10 08:13 storage
20 drwxr-xr-x root     root              2013-04-10 08:13 sys
21 drwxr-xr-x root     root              2012-12-31 03:20 system
22 -rw-r--r-- root     root          272 1970-01-01 00:00 ueventd.goldfish.rc
23 -rw-r--r-- root     root         4024 1970-01-01 00:00 ueventd.rc
24 lrwxrwxrwx root     root              2013-04-10 08:13 vendor -> /system/vendor
```

Listing 2.7: Android filesystem

The `/system` partition incorporates the entire Android operating system except the Linux kernel, which itself is located on the `/boot` partition. This folder contains the subdirectories `/system/bin` and `/system/lib` that contain core native executables and shared libraries correspondingly. Additionally, this partition encompass all system applications that are prebuilt with the system image. The image is mounted in read only mode (see Line 5 in Listing 2.5). Hence, the content of this partition cannot be changed

at runtime.

So as `/system` partition is mounted as read-only, it cannot be used for storing data. For this purposes the separate partition `/data` is allocated that responsible for storing user data or information changing over the time. For instance, `/data/app` directory contains all apk files of installed applications, while `/data/data` folder encloses "home" directories of the apps.

The `/cache` partition is responsible for storing frequently accessed data and application components. Additionally, the operating system over-the-air updates are also stored on this partition before being run.

So as `/system`, `/data` and `/cache` are formed during the compilation of Android, the default rights and owners to the files and folders contained on these images have to be defined at compile time. This means that the user and groups UIDs and GIDs should be available during the compilation of this operating system. The `android_filesystem_config.h` file (see Listing 2.8) contains the list of predefined users and groups. It should be mentioned that the values in some lines (for instance, see Line 10) correspond to the ones already defined on the *Linux Kernel* level, described in Section 2.3.

```
1  #define AID_ROOT             0 /* traditional unix root user */
2  #define AID_SYSTEM        1000 /* system server */
3  #define AID_RADIO         1001 /* telephony subsystem, RIL */
4  #define AID_BLUETOOTH     1002 /* bluetooth subsystem */
5  #define AID_GRAPHICS      1003 /* graphics devices */
6  #define AID_INPUT         1004 /* input devices */
7  #define AID_AUDIO         1005 /* audio devices */
8  #define AID_CAMERA        1006 /* camera devices */
9  ...
10 #define AID_INET          3003 /* can create AF_INET and AF_INET6 sockets */
11 ...
12 #define AID_APP          10000 /* first app user */
13 ...
14 static const struct android_id_info android_ids[] = {
15     { "root",       AID_ROOT, },
16     { "system",     AID_SYSTEM, },
17     { "radio",      AID_RADIO, },
18     { "bluetooth", AID_BLUETOOTH, },
19     { "graphics",  AID_GRAPHICS, },
20     { "input",      AID_INPUT, },
21     { "audio",      AID_AUDIO, },
22     { "camera",     AID_CAMERA, },
23 ...
24     { "inet",       AID_INET, },
25 ...
26 };
```

Listing 2.8: Android hard-coded UIDs and GIDs and their mapping to user names

Additionally, in this file the default rights, owners and owner groups of the files and folders are defined (see Listing 2.9). These rules are parsed and applied by `fs_config()` function, which is defined in the end of this file. This function is called during the assembly of the images.

```
1  /* Rules for directories.*/
2  static struct fs_path_config android_dirs[] = {
3      { 00770, AID_SYSTEM, AID_CACHE,  "cache" },
4      { 00771, AID_SYSTEM, AID_SYSTEM, "data/app" },
5  ...
6      { 00777, AID_ROOT,   AID_ROOT,   "sdcard" },
7      { 00755, AID_ROOT,   AID_ROOT,   0 },
8  };
9
10 /* Rules for files.*/
11 static struct fs_path_config android_files[] = {
12 ...
13     { 00644, AID_SYSTEM,     AID_SYSTEM,     "data/app/*" },
14     { 00644, AID_MEDIA_RW,   AID_MEDIA_RW,   "data/media/*" },
15     { 00644, AID_SYSTEM,     AID_SYSTEM,     "data/app-private/*" },
16     { 00644, AID_APP,        AID_APP,        "data/data/*" },
17     ...
18     { 02755, AID_ROOT,       AID_NET_RAW,    "system/bin/ping" },
19     { 02750, AID_ROOT,       AID_INET,       "system/bin/netcfg" },
20     ...
21     { 06755, AID_ROOT,       AID_ROOT,       "system/xbin/su" },
22     ...
23     { 06750, AID_ROOT,       AID_SHELL,      "system/bin/run-as" },
24     { 00755, AID_ROOT,       AID_SHELL,      "system/bin/*" },
25     ...
26     { 00644, AID_ROOT,       AID_ROOT,       0 },
27 };
```

Listing 2.9: Default permissions and owners

2.4.3 Native Executables Protection

It can be mentioned in Linsting 2.9 that some binaries are assigned with *setuid* and *setgid* access rights flags. For instance, the *su* program has them set. This well-known utility allows a user to run a program with the specified UID and GID. In Linux this functionality is usually used to run programs with superuser privileges. According to Listing 2.9, the binary `system/xbin/su` is assigned with the access rights equal to "`06755`" (see Line 21). The first non-zero number "`6`" means that this binary has *setuid* and *setgid* (`4 + 2`) access rights flags set. Usually, in Linux an executable is run with the same privileges as the process that has started it. These flags allows a user to run a program with the privileges of executable's owner or group [57]. Thus, in our case the binary `system/xbin/su` will be run as *root* user. These *root* privileges allow the program to change its UID and GID to the ones specified by a user (see Line 15 in Listing 2.10). After that, *su* may start the provided program (for instance, see Line 22) with the specified UID and GID. Therefore, the program will be started with the required UID and GID.

In the case of privileged programs it is required to restrict the circle of applications that have access to such utilities. In our case, without such restrictions any app may run *su* program and obtain *root* level privileges. In Android, such restrictions on the *Native Userspace level* are implemented comparing the UID of the calling program with the list of the UIDs allowed to run it. Thus, in Line 9 the *su* executable obtains the current

UID of the process, which is equal to the UID of the process calling it, and in Line 10 it compares this UID with the predefined list of allowed UIDs. Therefore, only if the UID of the calling process is equal to `AID_ROOT` or `AID_SHELL` the *su* utility will be started. To perform such check, *su* imports the UID constants (see Line 1) defined in Android.

```
1  #include <private/android_filesystem_config.h>
2  ...
3  int main(int argc, char **argv)
4  {
5      struct passwd *pw;
6      int uid, gid, myuid;
7
8      /* Until we have something better, only root and the shell can use su. */
9      myuid = getuid();
10     if (myuid != AID_ROOT && myuid != AID_SHELL) {
11         fprintf(stderr,"su: uid %d not allowed to su\n", myuid);
12         return 1;
13     }
14     ...
15     if(setgid(gid) || setuid(uid)) {
16         fprintf(stderr,"su: permission denied\n");
17         return 1;
18     }
19
20     /* User specified command for exec. */
21     if (argc == 3 ) {
22         if (execlp(argv[2], argv[2], NULL) < 0) {
23             fprintf(stderr, "su: exec failed for %s Error:%s\n", argv[2],
24                     strerror(errno));
25             return -errno;
26         }
27     ...
28 }
```

Listing 2.10: Source code of *su* program

Additionally, in newer versions (starting from 4.3) the Android core developers started to use *Capabilities* Linux kernel system [15]. This allows them additionally restrict the privileges of the programs that are required to run with *root* privileges. For instance, in the considered case of the *su* program it is not required to have all privileges of the *root* user. For this program it is enough only to have a possibility to change current UID and GID. Therefore, this utility requires only `CAP_SETUID` and `CAP_SETGID` root capabilities to operate correctly.

2.5 Android Security on the Application Framework Level

As we described in Section 2.2 the security on the *Application Framework* level is enforced by *IPC Reference Monitor*. In Section 2.5.1 we start our consideration of the security mechanisms on this level from the description of the inter-process communication system used in Android. After that we introduce *permissions* in Section 2.5.2, while in Section 2.5.3 we describe the permission enforcement system implemented on this level.

2.5.1 Android Binder Framework

As we described in Section 2.3.1, all Android applications are run in *Application Sandboxes*. Roughly saying, the sandboxing of the apps is provisioned by running all apps in different processes with different Linux identities. Additionally, system services are also run in separate processes with more privileged identities that allow them to get access to different parts of the system protected using the Linux Kernel DAC capabilities (see Sections 2.3 and 2.4). Thus, an Inter-Process Communication (IPC) framework is required to organize data and signals exchange between different processes. In Android, a special framework called *Binder* is used for inter-process communication [80]. The standard Posix *System V IPC* framework is not supported [1] by the Android implementation of the *Bionic libc* library. Moreover, additionally to the *Binder* framework for some special cases Unix domain sockets are used (e.g., for communication with the *Zygote* daemon) but the consideration of these mechanisms is out of the scope of this work.

The *Binder* framework was specifically redeveloped to be used in Android. It provides the capabilities required to organize all types of communication between processes in this operating system. Basically, even the mechanisms, such as *Intents* and *ContentProviders*, well-known to application developers, are built on top of the *Binder* framework. This framework provides the variety of features, such as the possibility to invoke methods on remote objects as if they were local, synchronous and asynchronous method invocation, *link to death* [2], ability to send file descriptors across processes, etc. [80, 127].

The communication between the processes is organized according to synchronous client-server model. The client initiates a connection and waits for a reply from the server side. Thus, the communication between the client and the server may be imagined as they are executed in the same process thread. This provides a developer with the possibility to invoke methods on remote objects as if they were local. The communication model through Binder is presented in Figure 2.5. In this figure, the application in *Process A*, which acts as a Client, wants to use the behavior exposed by a Service, which runs in *Process B* [80].

All communications between clients and services using the *Binder* framework happens through a Linux kernel driver `/dev/binder`. The permissions to this device driver is set to world readable and writable (see Line 3 in Listing 2.6 located in Section 2.4.1). Hence, any application may write to and read from this device. To conceal the peculiarities of the *Binder* communication protocol, the *libbinder* library is used in Android. It provides the facilities to make the process of interaction with the kernel driver transparent for an app developer. In particular, all communications between a Client and a Server happen through proxies on the client side and stubs on the server side. The proxies and the stubs are responsible for marshaling and unmarshaling the data and the commands sent

[1] `https://android.googlesource.com/platform/ndk/+/android-4.2.2_r1.2/docs/system/libc/SYSV-IPC.html`

[2] Link to Death is an automatic notification when a Binder of a certain process is terminated

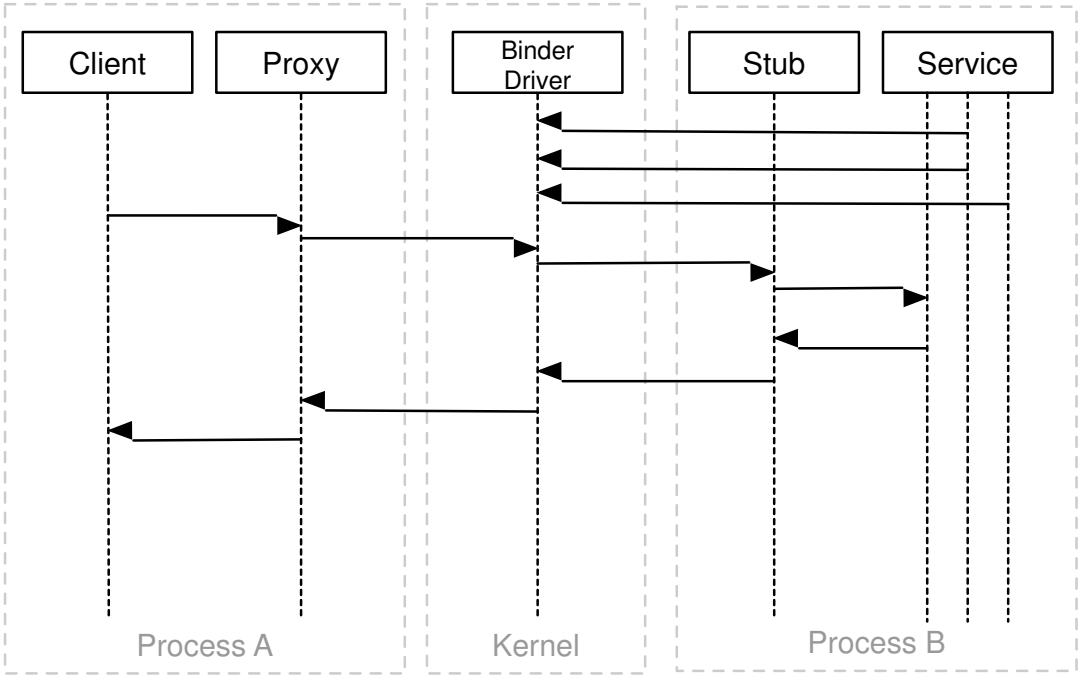


Figure 2.5: Android Binder communication model [80]

over the *Binder* driver. To make use of proxies and stubs a developer just defines an *AIDL* interface that is transformed into a proxy and a stub during the compilation of the application. On the server side, a separate Binder thread is invoked to process a client request.

Technically, each Service (sometimes called as *Binder Service*) exposed using the *Binder* mechanism is assigned with a token. The kernel driver ensures that this 32 bit value is unique across all processes in the system. Thus, this token is used as a handle to a *Binder Service*. Having this handle it is possible to interact with the Service. However, to start using the Service the Client at first has to discover this value. The discovery of Service's handle occurs using Binder's *context manager* (*servicemanager* is Android's implementation of Binder's *context manager*. Here we use these notions interchangeably). *Context manager* is a special *Binder Service* with the predefined handle value equal to `0` (the reference to which is obtained in Line 8 in Listing 2.11). So as it has a fixed handle value, any party can find it and call its methods. Basically, *context manager* acts as a name service providing the handle of a Service using the name of this Service. To achieve this goal, each Service must be registered with *context manager* (for instance, using the method `addService` of the `ServiceManager` class in Line 26). Thus, a Client has to know only the name of a Service to communicate with it. Resolving this name using *context manager* (see method `getService` Line 12) the Client receives the token that is later used for the interactions with the Service. The *Binder* driver allows only a single *context*

manager to be registered. Therefore, *servicemanager* is one of the first services started by Android (see Section 2.4.1). The component *servicemanager* ensures that only the privileged system identities are allowed to register services.

```
public final class ServiceManager {
    ...
    private static IServiceManager getIServiceManager() {
        if (sServiceManager != null) {
            return sServiceManager;
        }
        // Find the service manager
        sServiceManager = ServiceManagerNative.asInterface(BinderInternal.getContextObject());
        return sServiceManager;
    }

    public static IBinder getService(String name) {
        try {
            IBinder service = sCache.get(name);
            if (service != null) {
                return service;
            } else {
                return getIServiceManager().getService(name);
            }
        } catch (RemoteException e) {
            Log.e(TAG, "error in getService", e);
        }
        return null;
    }

    public static void addService(String name, IBinder service, boolean allowIsolated) {
        try {
            getIServiceManager().addService(name, service, allowIsolated);
        } catch (RemoteException e) {
            Log.e(TAG, "error in addService", e);
        }
    }
    ...
}
```

Listing 2.11: The sources of `ServiceManager`

The *Binder* framework does not impose any security by itself. At the same time, it provides the facilities to procure the security in Android. The *Binder* driver adds the UID and the PID of the sender process to each transaction. So as each application in the system has its own UID, this value may be used to identify the calling party. The receiver of the call may check the obtained values and decide if the transaction should be completed. The receiver may get the UID and the PID of the sender using the calls `android.os.Binder.getCallingUid()` and `android.os.Binder.getCallingPid()` [80]. Additionally, a *Binder* handle may also act as a security token due to its uniqueness across all the processes and the obscurity of its value [108].

2.5.2 Android Permissions

As we consider in Section 2.3.1, in Android each application by default obtains its own UID and GID system identities. Additionally, there are also a number of the identities hardcoded in the operating system (see Listing 2.8). These identities are used to separate the components of the Android operating system using the *DAC* enforced on the *Linux Kernel* level, thus, increasing the overall security of the operating system. Among these identities `AID_SYSTEM` stands out. This UID is used to run the *System Server* (`system_server`), the component that unites the services provided by the Android OS. The *System Server* has a privileged access to the operating system resources, and each service run within the *System Server* provides the controlled access to a particular functionality to other OS components and applications. This controlled access is backed by the *permission* system.

As we consider in Section 2.5.1, the *Binder* framework provides the ability to get the UID and the PID of the sender on the receiver side. In general case, this functionality may be exploited by a service to control consumers that want to connect to the service. This can be achieved by comparing the UID and/or PID of a consumer with the list of UIDs allowed by the service. However, in Android this functionality is implemented in a slightly different manner. Each critical functionality of a service (or simply saying a method of a service) is guarded with a special label called *permission*. Roughly saying, before running such method a check if the calling process is assigned with the *permission*, is performed. If the calling process has the required permission then the service invocation will be allowed. Otherwise, a security check exception will be thrown (usually, `SecurityException`). For instance, if a developer wants to provide her app with a possibility to send SMS she has to add into app's `AndroidManifest.xml` file the following line `<uses-permission android:name="android.permission.SEND_SMS" />`. Android also provides a set of special calls that allow to check at runtime if a service consumer has been assigned with a permission.

The permission model described so far provides an effective way to enforce security. At the same time, this model is ineffective because it considers all the permissions as equal. At the same time, in the case of mobile operating systems the provided capabilities may not be always equal in the security sense. For instance, the capability to install applications is more critical then the ability to send SMSes, which in turn is more dangerous then the setting an alarm or vibrating.

This problem is addressed in Android by introducing the security levels of permissions. There are four possible levels of permissions: `normal`, `dangerous`, `signature` and `signatureOrSystem`. The level of permissions is either hardcoded into the Android operating system (for system permissions) or assigned by a developer of a third-party app in the declaration of a custom permission. This level influences on a decision whether to

grant the permission to a requesting application. To be granted, the `normal` permissions have to be just requested in application's `AndroidManifest.xml` file. The `dangerous` permissions, besides to be requested in the manifest file, have to be also approved by a user. In this case, during the installation of an app the user is displayed with the set of permissions requested by the package. If the user approves them, then the application will be installed. Otherwise, the installation will be canceled. The `signature` permission is granted by the system if the app requested the permission is signed with the same signatures as the application that has declared it (the usage of app signatures in Android is considered in Section 2.6.3). The `signatureOrSystem` permission is granted either if the apps requesting and the declaring the permission are signed with the same certificates or the requesting application is located on the system image. Thus, for our example the vibrating capability will be protected with the permission of the `normal` level, send SMSes functionality will be guarded with the `dangerous` permission level and package installation ability will be secured with the `signatureOrSystem` permission level.

Definition of System Permissions

System permissions, which are used to protect Android operating system functionality, are defined in framework's `AndroidManifest.xml` file located in `frameworks/base/core/res` folder of the Android sources. An excerpt of this file with several permission definition examples is shown in Listing 2.12. In these examples the permission declarations are shown used to protect sending SMSes, vibrator and package installation functionality.

By default the developers of third-party applications do not have access to the functionality protected with system permissions of levels `signature` and `signatureOrSystem`. This behaviour is ensured in the following way. The *Application Framework* package is signed with the *platform* certificate. Thus, the applications requiring the functionality protected with the permissions of these levels must be signed with the same *platform* certificate. However, the access to the private key of this certificate is available only to the builders of the operating system, usually hardware producers (who make their own customization of Android) or telecom operators (who distribute the phones with their modified images of operating systems).

Permission Management

The system service `PackageManagerService` is responsible for the application management in Android. This service assists the installation, uninstallation and update of applications in the operating system. Another important role of this service is permission management. Basically, it can be considered as a policy administration point. It stores the information that allows to check if an Android package is assigned with a particular permission. Additionally, during the installation and upgrade of applications it performs

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="android" coreApp="true" android:sharedUserId="android.uid.system"
    android:sharedUserLabel="@string/android_system_label">
    ...
    <!-- Allows an application to send SMS messages. -->
    <permission android:name="android.permission.SEND_SMS"
        android:permissionGroup="android.permission-group.MESSAGES"
        android:protectionLevel="dangerous"
        android:permissionFlags="costsMoney"
        android:label="@string/permlab_sendSms"
        android:description="@string/permdesc_sendSms" />
    ...
    <!-- Allows access to the vibrator -->
    <permission android:name="android.permission.VIBRATE"
        android:permissionGroup="android.permission-group.AFFECTS_BATTERY"
        android:protectionLevel="normal"
        android:label="@string/permlab_vibrate"
        android:description="@string/permdesc_vibrate" />
    ...
    <!-- Allows an application to install packages. -->
    <permission android:name="android.permission.INSTALL_PACKAGES"
        android:label="@string/permlab_installPackages"
        android:description="@string/permdesc_installPackages"
        android:protectionLevel="signature|system" />
    ...
</manifest>
```

Listing 2.12: The definitions of system permissions

a bunch of checks to ensure that the integrity of permission model is not violated during these processes. Moreover, it also acts as a policy decision point. The methods of this service (as we will show later) are the last elements in the chain of the permission checks. We will not consider the operation of `PackageManagerService` here. However, the interested reader may refer to [115, 144] to get some more details how the installation of applications is performed.

`PackageManagerService` stores all information related to permissions of third-party applications in the `/data/system/packages.xml` [33]. This file is used as a persistent storage between the restarts of the system. However, at runtime all information about permissions is preserved in RAM allowing to increase the responsiveness of the system. This information is collected during the boot using the data stored in the `packages.xml` file for third-party applications and through parsing system apps.

2.5.3 Permission Enforcement on the Application Framework level

To understand how Android enforces permissions on the *Application Framework* level, for instance, let consider the *Vibrator Service*. In Listing 2.13 in Line 6 an example how the *Vibrator Service* protects its method `vibrate` is shown. In this line the check is performed if a calling component is assigned with the label `android.permission.VIBRATE` defined by the constant `android.Manifest.permission.VIBRATE`. Android provides several methods to check if a sender (or service consumer) has been assigned with a permission. In

our case, these facilities are represented by the method `checkCallingOrSelfPermission`. Additionally to this method, there are also a number of other methods that can be used to check the permissions of the service caller.

```
1  public class VibratorService extends IVibratorService.Stub
2          implements InputManager.InputDeviceListener {
3      ...
4      public void vibrate(long milliseconds, IBinder token) {
5          if (mContext.checkCallingOrSelfPermission(android.Manifest.permission.VIBRATE)
6                  != PackageManager.PERMISSION_GRANTED) {
7              throw new SecurityException("Requires VIBRATE permission");
8          }
9          ...
10     }
11     ...
12 }
```

Listing 2.13: The check of a permission

The implementation of the method `checkCallingOrSelfPermission` is shown in Listing 2.14. In Line 24 the method `checkPermission` is called. It takes the `uid` and the `pid` as parameters that are provided by the *Binder* framework.

```
1  class ContextImpl extends Context {
2      ...
3      @Override
4      public int checkPermission(String permission, int pid, int uid) {
5          if (permission == null) {
6              throw new IllegalArgumentException("permission is null");
7          }
8
9          try {
10             return ActivityManagerNative.getDefault().checkPermission(
11                     permission, pid, uid);
12         } catch (RemoteException e) {
13             return PackageManager.PERMISSION_DENIED;
14         }
15     }
16
17     @Override
18     public int checkCallingOrSelfPermission(String permission) {
19         if (permission == null) {
20             throw new IllegalArgumentException("permission is null");
21         }
22
23         return checkPermission(permission, Binder.getCallingPid(),
24                 Binder.getCallingUid());
25     }
26     ...
27 }
```

Listing 2.14: The excerpt of `ContextImpl` class

In Line 11, the check is redirected to the `ActivityManagerService` class that in turn performs the actual check in the method `checkComponentPermission` of the `ActivityManager` component. The code of this method is presented in Listing 2.15. In Line 4 it checks if the caller UID belongs to the privileged ones. The components with the *root* and *system*

UIDs are granted by the system with all permissions.

```
1  public static int checkComponentPermission(String permission, int uid,
2          int owningUid, boolean exported) {
3      // Root, system server get to do everything.
4      if (uid == 0 || uid == Process.SYSTEM_UID) {
5          return PackageManager.PERMISSION_GRANTED;
6      }
7      // Isolated processes don't get any permissions.
8      if (UserId.isIsolated(uid)) {
9          return PackageManager.PERMISSION_DENIED;
10     }
11     // If there is a uid that owns whatever is being accessed, it has
12     // blanket access to it regardless of the permissions it requires.
13     if (owningUid >= 0 && UserId.isSameApp(uid, owningUid)) {
14         return PackageManager.PERMISSION_GRANTED;
15     }
16     // If the target is not exported, then nobody else can get to it.
17     if (!exported) {
18         Slog.w(TAG, "Permission denied: checkComponentPermission() owningUid=" + owningUid);
19         return PackageManager.PERMISSION_DENIED;
20     }
21     if (permission == null) {
22         return PackageManager.PERMISSION_GRANTED;
23     }
24     try {
25         return AppGlobals.getPackageManager()
26                 .checkUidPermission(permission, uid);
27     } catch (RemoteException e) {
28         // Should never happen, but if it does... deny!
29         Slog.e(TAG, "PackageManager is dead?!?", e);
30     }
31     return PackageManager.PERMISSION_DENIED;
32 }
```

Listing 2.15: The sources of the method `checkComponentPermission` of the `ActivityManager`

In Line 26 in Listing 2.15 the permission check is redirected to *Package Manager* that forwards it to `PackageManagerService`. As we explained before, this service knows what permissions are assigned to Android packages. The `PackageManagerService` method, which performs the permission check, is presented in Listing 2.16. In Line 7 the exact check is performed if a permission is granted to the Android app defined by its UID.

2.6 Android Security on the Application Level

Although in this section we describe the security on the *Application* level, the actual security enforcement usually happens on lower layers described so far. However, it is easier to explain some security features of Android after introducing the *Application* level.

2.6.1 Application Components

Android apps are distributed in the form of Android Package (.apk) files. A package consists of Dalvik executable files, resources files, a manifest file and native libraries, and is signed by the developer of the applications using self-signed certificate.

```
1  public int checkUidPermission(String permName, int uid) {
2      final boolean enforcedDefault = isPermissionEnforcedDefault(permName);
3      synchronized (mPackages) {
4          Object obj = mSettings.getUserIdLPr(UserHandle.getAppId(uid));
5          if (obj != null) {
6              GrantedPermissions gp = (GrantedPermissions)obj;
7              if (gp.grantedPermissions.contains(permName)) {
8                  return PackageManager.PERMISSION_GRANTED;
9              }
10         } else {
11             HashSet<String> perms = mSystemPermissions.get(uid);
12             if (perms != null && perms.contains(permName)) {
13                 return PackageManager.PERMISSION_GRANTED;
14             }
15         }
16         if (!isPermissionEnforcedLocked(permName, enforcedDefault)) {
17             return PackageManager.PERMISSION_GRANTED;
18         }
19     }
20     return PackageManager.PERMISSION_DENIED;
21 }
```

Listing 2.16: The sources of method `checkUidPermission` of `PackageManagerService`

Each Android application consists of several components of four component types: *Activities*, *Services*, *Broadcast Receivers* and *Content Providers*. The separation of an application into the components supports the reuse of application parts between the apps.

- **Activity**. An *Activity* is an element of user interface. Generally speaking, the activity often represents a screen.
- **Service**. A *Service* is a background worker in Android. The service can run indefinite time. The most famous example of a service is media player that plays music in the background even if the user leaves the activity that has started this service.
- **Broadcast receiver**. A *Broadcast Receiver* is a component of an application that receives broadcast messages and starts a workflow according to the obtained message.
- **Content provider**. A *Content Provider* is a component that provides an application with abilities to store and retrieve data. It also permits to share a set of data with another application.

So as Android applications consist of different components, there is no central entry point unlike Java programs with the `main` method. Having no central point, all components (with an exception to broadcast receivers that may also be defined dynamically) need to be declared by the developer of an application in the `AndroidManifest.xml` file. The separation into components makes possible to use parts in other applications. For instance, in Listing 2.17 an example of app's `AndroidManifest.xml` file is shown. This application consists of one *Activity* declared in Line 21. Other applications may call this activity integrating the functionality of this component into their apps.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.testpackage.testapp"
    android:versionCode="1"
    android:versionName="1.0"
    android:sharedUserId="com.testpackage.shareduid"
    android:sharedUserLabel="@string/sharedUserId" >

    <uses-sdk android:minSdkVersion="10" />

    <permission android:name="com.testpackage.permission.mypermission"
        android:label="@string/mypermission_string"
        android:description="@string/mypermission_descr_string"
        android:protectionLevel="dangerous" />

    <uses-permission android:name="android.permission.SEND_SMS"/>

    <application
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name" >
        <activity android:name=".TestActivity"
            android:label="@string/app_name"
            android:permission="com.testpackage.permission.mypermission" >
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
            <intent-filter >
                <action android:name="com.testpackage.testapp.MY_ACTION" />
                <category android:name="android.intent.category.DEFAULT" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Listing 2.17: Example of the `AndroidManifest.xml` file

Android provides a variety of methods to invoke the components of applications. A new *Activity* is started by using the methods `startActivity` and `startActivityForResult`. *Services* are started through the `startService` method. In this case, called service invokes its method `onStart`. When a developer is going to establish a connection between a component and a service she invokes the `bindService` method and the `onBind` method is invoked in the called service. *Broadcast receivers* are started when an app or system component send special messages using the methods `sendBroadcast`, `sendOrderedBroadcast` and `sendStickyBroadcast`.

Content providers are invoked by the requests from content resolvers. All other component types are activated through *Intents*. *Intents* is a special mean of communication in Android based on the *Binder* framework. Intents are passed into the methods that perform component invocation. The called component can be invoked by two different types of intents. To show the differences of these types, let consider an example. For instance, a user wants to choose a picture in an application. The developer of the application can use an *Explicit Intent* or an *Implicit Intent* to invoke a component that selects a picture. For the first intent type, the developer realizes picking functionality in the component of his

application and calls this component using the *Component Name* data field of the explicit intent. Of course, the developer can invoke a component of other application, but, in this case, he has to be sure that this application is installed in the system. Generally, from the developer's point of view, there is no difference between the interactions of components inside one application or among components of different applications. For the second intent type, the developer transfers the right to choose the appropriate component to the operating system. The intent object contains some information in its *Action*, *Data* and *Category* fields. According to this information, using *Intent Filters* the operating system chooses the proper component that may process the intent. An intent filter defines the "template" of intents the component can process. Of course, the same application can define an intent filter that will process intents from other component.

2.6.2 Permissions on the Application Level

Permissions are used not only for protecting the access to the system resources. The developers of third-party applications may also use custom permissions to guard the access to the components of their applications. An example of custom permission declaration is shown in Listing 2.17 in Line 11. The declaration of custom permissions is similar to the one of the system permissions.

To illustrate the usage of custom permissions let refer to Figure 2.6. The *Application 2* consisting of 3 components wants to protect the access to two of them: $C1$ and $C2$. To achieve this goal the developer of the *Application 2* has to declare two permission labels $p1$, $p2$ and assign them to protected components correspondingly. If a developer of the *Application 1* wants ot obtain access to component $C1$ of the *Application 2* she must define that her app requires permission $p1$. In this case, the *Application 1* receives a possibility to use the component $C1$ of the *Application 2*. If the app has not specified the required permission, the access to the component guarded with this permission is prohibited (see the case of the component $C2$ in Figure 2.6). Referring back to our example of the `AndroidManifest.xml` file in Listing 2.17, the activity `TestActivity` is protected with the permission `com.testpackage.permission.mypermission`, which is declared in the same application manifest file. If another application wants to use the functionality provided by `TestActivity`, it must request the usage of this permission, similarly to how it is done in Line 16.

`ActivityManagerService` is responsible for the invocation of the components of application. To enforce the security of app components, in the framework methods (e.g., `startActivity` described in Section 2.6.1), which are used to invoke the components, the special hooks are placed. These hooks check if an application has permission to call the component. These checks end with the `checkUidPermission` method of `PackageManagerServer` (see Listing 2.16). Thus, the actual permission enforcement happens on the *Application*

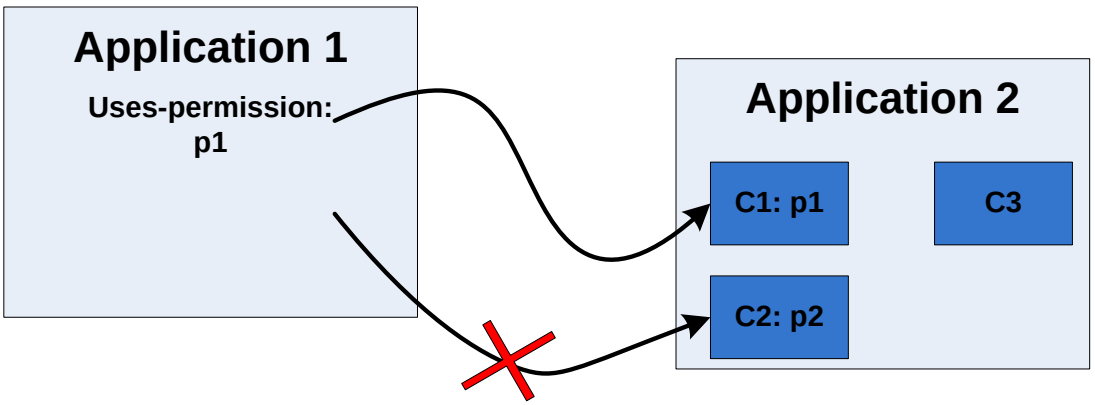


Figure 2.6: Permission enforcement to guard the components of third-party applications

Framework level that is considered as a trusted part of the Android operating system. Hence, the check cannot be bypassed by applications. More information about how the components are called and permission checks can be found in [37].

2.6.3 Application Signing Process

Android applications are spread across the devices in the form of Android *Application Package files* (`apk` files). As programs for this platform are mainly written in Java, not surprisingly this format has a lot in common with the Java packaging format – `jar` (Java ARchive), which is used to combine code, resource and metadata (from an optional `META-INF` directory) files into one file using the `zip` archiving algorithm. The META-INF directory stores package and extension configuration data, including security, versioning, extension and services [23]. Basically, in the case of Android the *apkbuilder* tool zips together built project files [4] and then this archive is signed with the standard Java utility *jarsigner* [24]. During the application signing process `jarsigner` creates the `META-INF` directory that usually contains the following files in case of Android: *manifest file* (`MANIFEST.MF`), *signature files* (with `.SF` extension) and *signature block files* (`.RSA` or `.DSA`).

The *manifest file* (`MANIFEST.MF`) consists of the main attributes section and per-entry attributes, one entry for each file contained in the unsigned apk. These per-entry attributes store information about the file name and a digest of the file contents encoded using the `base64` format. On Android, the `SHA1` algorithm is used to compute the digest. An excerpt from a manifest file is presented in Protocol 1.

The content of the *signature file* (`.SF`), which contains data to be singed, is similar to the one of `MANIFEST.MF`. An example of this file is presented in Protocol 2. Main section contains a digest (`SHA1-Digest-Manifest-Main-Attributes`) of the main attributes and a digest (`SHA1-Digest-Manifest`) of the content of the manifest file. Per-entry section

Protocol 1 An excerpt from a manifest file.

```
Manifest-Version: 1.0
Created-By: 1.6.0_41 (Sun Microsystems Inc.)

Name: res/layout/main.xml
SHA1-Digest: NJ1YLN3mBEKTPibVXbFO8eRCAr8=

Name: AndroidManifest.xml
SHA1-Digest: wBoSXxhOQ2LR/pJY7Bczu1sWLy4=
```

contains digests of entries in the manifest file with the corresponding file names.

Protocol 2 An excerpt from a signature file.

```
Signature-Version: 1.0
SHA1-Digest-Manifest-Main-Attributes: nl/DtR972nRpjey6ocvNKvmjvw8=
Created-By: 1.6.0_41 (Sun Microsystems Inc.)
SHA1-Digest-Manifest: Ej5guqx3DYaOLOm3Kh89ddgEJW4=

Name: res/layout/main.xml
SHA1-Digest: Z871jZHrhRKHDaGf2K4p4fKgztk=

Name: AndroidManifest.xml
SHA1-Digest: hQtlGk+tKFLSXufjNaTwd9qd4Cw=
...
```

The last part in the chain is the *signature block file* (`.DSA` or `.RSA`). This binary file contains a signed version of the signature file; it has the same name as the corresponding `.SF` file. Depending on the used algorithm (RSA or DSA) it has different extensions.

It is possible to sign the same apk file with several different certificates. In this case in the `META-INF` directory there will be several `.SF` and `.DSA` or `.RSA` files (their number will be equal to the number of times the application was signed).

App Signature Check in Android

Most of Android apps are sealed with a developer-signed certificate (notice that for Android "certificate" and "signature" can be used interchangeably). This certificate is used for assurance that the code of the original application and its update come from the same place, and to establish trust relationships between applications of the same developer. To perform this check Android simply compares binary representations of certificates, which were used to sign an application and its update (in the first case) and collaborating applications (in the second).

This check of certificates is implemented in `PackageManagerService` by the method

`int compareSignatures(Signature[] s1, Signature[] s2)`, which code is presented in Listing 2.18. In the previous section we noted that in Android it is possible to sign the same application with several different certificates. This explains why the method takes two arrays of signatures as parameters. Despite the fact that this method takes the central place in the Android security provision, its behaviour strongly depends on the version of the platform. In the newer versions (starting from Android 2.2) this method compares two arrays of `Signature`, and if both arrays are not equal to `null` returns `SIGNATURE_MATCH` value if all `s2` signatures are contained in `s1`, and `SIGNATURE_NO_MATCH` otherwise. Before the version 2.2, this method checked if array `s1` is contained in `s2`. That behaviour allowed the system to install upgrades even if they had been signed only with a subset of certificates of the original application [5].

```
1  public class PackageManagerService extends IPackageManager.Stub {
2      ...
3      static int compareSignatures(Signature[] s1, Signature[] s2) {
4          if (s1 == null) {
5              return s2 == null
6                      ? PackageManager.SIGNATURE_NEITHER_SIGNED
7                      : PackageManager.SIGNATURE_FIRST_NOT_SIGNED;
8          }
9          if (s2 == null) {
10             return PackageManager.SIGNATURE_SECOND_NOT_SIGNED;
11         }
12         HashSet<Signature> set1 = new HashSet<Signature>();
13         for (Signature sig : s1) {
14             set1.add(sig);
15         }
16         HashSet<Signature> set2 = new HashSet<Signature>();
17         for (Signature sig : s2) {
18             set2.add(sig);
19         }
20         // Make sure s2 contains all signatures in s1.
21         if (set1.equals(set2)) {
22             return PackageManager.SIGNATURE_MATCH;
23         }
24         return PackageManager.SIGNATURE_NO_MATCH;
25     }
26     ...
27 }
```

Listing 2.18: The sources of method `compareSignatures`

Trust relationships between applications of the same developer are required in several cases. The first case is connected with the permissions of the levels `signature` and `signatureOrSystem`. To use the functionality protected with the permissions of these levels, the packages declaring the permission and requesting it must be signed with the same set of certificates. The second case related to Android's capability to run different applications with the same UID or even in the same Linux process. In this case, applications requested such behavior must be signed with the same signature.

Chapter 3

Fast Detection of Repackaged Android Applications

Mobile application repackaging on Android is known to be a source of revenue loss of the developers, as well as an avenue for malware distribution. The ease of Android applications repackaging and proliferation of application clones in Google Play and other markets call for new effective techniques to detect repackaged code and combat distribution of cloned applications. Today all existing techniques for repackaging detection are based on code similarity or feature (e.g., permission set) similarity evaluation. In this chapter we propose a new approach to detect repackaging based on the resource files available in application packages. Our tool called FSquaDRA performs a quick pairwise application comparison, as it measures how many identical resources are present inside both packages under analysis. The intuition behind our approach is that malicious repackaged applications still need to maintain the "look and feel" of the originals by including the same images and other resource files, even though they might have additional code included or some of the original code removed.

This chapter proceeds as follows. We introduce the problem of application repackaging and give short description of our solution in Section 3.1. We observe our approach and implementation details in Section 3.2, and the collected dataset in Section 3.3. Section 3.4 reports on the FSquaDRA performance, and details the comparison with AndroGuard. Section 3.5 provides an insight into Google Play apps repackaging rates, and on discovered clusters of very similar applications. Section 3.6 overviews the related work.

3.1 The Problem of Application Repackaging

Mobile ecosystems today represent a huge and fast growing market. Success stories of such companies as Rovio (with the Angry Birds game) attract to the mobile business vast amounts of developers. Yet, the developers can suffer from monetary and reputation

losses when their applications are stolen and appear on the markets *repackaged*.

The problem of application (app for short) stealing on Android stems from the fact that at present it is not very difficult to repackage an Android app. Applications are usually signed with a self-signed certificate. Thus, an adversary can easily change the code and sign the application with his own certificate. At present, neither the official Google Play market nor alternative markets do not detect if an application has been repackaged. At the same time, there is a strong aspiration from adversaries to steal applications. They can earn monetary profits either by changing the revenue destination of advertisement libraries, or by embedding malware, which can transform phones into controllable "zombies". Thus, to maintain the healthiness of the Android markets there is a strong need to detect the repackaged applications and prevent their distribution.

Currently the problem of Android application repackaging is widely explored and several solutions to identify plagiarized applications were proposed, e.g., [63, 64, 85, 88, 118, 154]. All these solutions are based on features extracted from the application code. However, it is clear that the code itself is often impacted by the repackaging process: the added malicious functionality (new advertisement libraries and/or malware code) modifies the code of the app. Additionally, the usage of obfuscation libraries during the repackaging (this possibility is explored in [96]) can further modify the code. Moreover, adversaries can simply replicate some initial behaviour of an application (so called app spoofing [137]). We can therefore conclude that the detection rates of repackaging for a code similarity-based techniques decrease under the influence of these factors. Notice that the availability of various tools like smali/backsmali [36] or apktool [3] greatly alleviates the task of code changing and application repackaging.

Yet, it is not only the code that defines an app. Nowadays, smartphones have powerful processors, advanced video and audio systems that are able to support screens with very high resolutions and to produce sounds of high quality. These factors lead to the constant demand of attractive applications. Therefore, to become popular an app should not only include the code with interesting functionality, but should also contain attractive layouts, images and other supplement resources, which become an integral part of user experience. Thus, the resource files are now inseparable part of modern mobile applications and sometimes their design requires more efforts then the process of code development, which is greatly facilitated now by the presence of large variety of different libraries. These resource files (resources, for short) are delivered on the device packaged together with the code.

In this chapter we propose an approach, which can be used to detect repackaged applications based on comparison of the content of the resource files forming Android application packages. Our approach relies on the observations that usually Android application packages (`apk` files) include a significant number of resource files, and that malicious repackagers aim to change the applications in a way they resemble the originals as much

as possible. Therefore, the code parts may change but the resource files (including icons, images, music and video files, etc.) often remain the same.

To be practical, the approach of detecting repackaged applications based on resource files comparison needs to be fast enough, considering the vast number of Android applications (currently there are more then 700,000 apps only in the official market). Thus, a simple pairwise comparison of all files inside two compared apps is not quite scalable because the complexity is proportional to the product of the number of files inside two packages multiplied by the average size of a file. We may reduce the complexity by computing hashes of files and comparing only the digests. However, this solution still requires computational resources to compute the hashes. Luckily, during the process of application signing a hash of each file inside the `apk` is computed and stored inside the package. We leverage this information to compute the similarity of applications. Thus, our approach is fast enough to be used even for comparing applications pairwise.

To our knowledge, we are the first who propose to detect Android repackaged applications based on similarities in resource files, and not on the ones in the code.

3.2 Our approach

Android applications are spread across the devices in the form of Android *Application Package Files* (`apk` files). As Android heavily relies on Java, it is not very surprising that `apk` files have a lot in common with the Java packages. Java ARchive files (`jar`) are used to combine code, resource and metadata into one file using the `zip` archiving algorithm. Similarly, Android packages contain code, manifest, libraries and resource files in a `zip` archive. Thus, each app includes not only the code, but also a large set of supplementary files being an integral part of the Android package. To confirm this we analysed the applications in our dataset. For our dataset, on average there are 315.56 files inside an Android application, where the maximum value constitutes 11099 files and minimum is 4. Figure 3.1 shows the distribution of the number of files inside an apk obtained during this experiment (in Figure 3.1 we limited the number of files to 2000 because there is a small fraction of applications with more than 2000 files).

Previously, to detect repackaged applications the researchers considered predominantly the code (`classes.dex`) and the manifest `AndroidManifest.xml` files. We propose to use the full set of file inside apks to detect repackaging.

Our intuitions are as follows. An adversary, who clones an application, seeks to resemble the original one as much as possible, thus, increasing the probability of the clone installation. In Android apps code is loosely coupled with resources giving the adversary a possibility to easily change the code. For example, the legitimate Opera Mini application and its repackaged version containing malware [119] coincide in 230 out of 234 files inside those packages. The only different files are: `resources.arsc`,

`AndroidManifest.xml`, `classes.dex` and `res/layout/main.xml`. Clearly, the modifications in `AndroidManifest.xml` are required to obtain new permissions for the malware to operate, while `classes.dex` was modified to embed the actual malicious code.

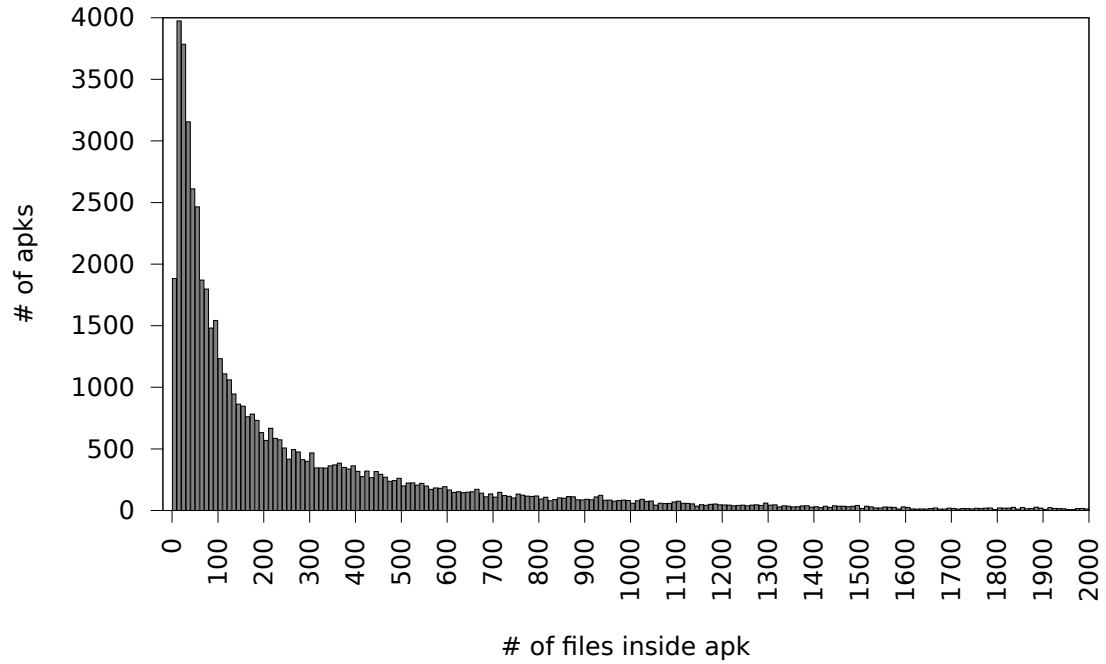


Figure 3.1: Distribution of number of files inside apk

For the scope of this work we consider two cases of repackaging: (malicious) *plagiarism*, when two application packages include the same files but are signed by different developers (with different certificates), and (benign) *rebranding*, when two application packages include the same files and are signed by the same certificate.

Using binary comparison of files, which constitute two Android applications, it is possible to understand to what extent these two apps are similar. Unfortunately, binary comparison is not a cheap operation. Moreover, a file in the first app should be compared against each file in the second package. These overheads may be considerably reduced using comparison of the file digests (hashes). Our tool uses this technique to calculate the similarity between two applications. At the same time, digest computation against the content of a file requires considerable resources consumption and, thus, directly cannot be used in a tool that has to process significant amount of apks. To overcome this limitation we use the hashes calculated during the application signing process (see Section 2.6.3). Thus, the overhead for hash computations does not affect our tool.

3.2.1 The algorithm and implementation details

In this section we report on the algorithm and implementation details of FSquaDRA. Protocol 3 describes the algorithm implemented in FSquaDRA for pairwise comparison of all applications located under a specific directory provided as an argument to our tool. Additionally, FSquaDRA has options of comparing just two apps, an application and all packages under a specified folder or applications in two folders. The implementation slightly changes in these cases but the main idea remains the same.

Protocol 3 The algorithm of application comparison

```
1: ApkAttr_list ← [ ]
2: Apk_list ← getApkFileList(path)
3:
4: \\ Get application attributes:
5: \\ certificates, relative file paths and hashes
6: for all A_i ∈ Apk_list do
7:     ApkName_i ← getApkName(A_i)
8:     Attr_i ← getApkAttributesToMemory(A_i)
9:     Add (fileName_i, Attr_i) to ApkAttr_list
10: end for
11:
12: size ← length(ApkAttr_list)
13:
14: \\ Pairwise comparison of applications
15: for (k = 0; k < size; k++) do
16:     hashes_k ← getFileHashesSet(Attr_k)
17:     certs_k ← getCertHashes(Attr_k)
18:     for (l = k + 1; l < size; l++) do
19:         hashes_l ← getFileHashesSet(Attr_l)
20:         certs_l ← getCertHashes(Attr_l)
21:         jSim ← getJaccardIndex(hashes_k, hashes_l)
22:         sameCert ← certsTheSame(certs_k, certs_l)
23:         OUT: ApkName_k, ApkName_l, sameCert, jSim
24:     end for
25: end for
```

To begin with, we list all the apk files that need to be considered during the analysis. In Line 2 of Protocol 3 we select all apk files located under the directory, the path to which is specified by the variable *path* provided as an argument to our tool. After that, in Lines 6-10 FSquaDRA extracts the required information from the apk files. At first, our tool gets the name of the file. Then it extracts the attributes of the apk using the `getApkAttributesToMemory` method. In particular, it iterates over the entries in the `MANIFEST.MF` file and writes the results into a map, which key corresponds to the relative path of a file inside the package and value is equal to the SHA1 hash of the file. Additionally, during this step FSquaDRA extracts the developer certificates, which have been used for the application signing, and stores into *Attr* object the digests computed over these certificates. This allows us to reduce the memory consumption of FSquaDRA and speed up the certificate comparison process. The name of the app file along with the object $Attr_i$ containing all required application attributes are stored into the $ApkAttr_{list}$ list.

Lines 15-25 show how the comparison of applications is performed. This comparison consists of two main steps, namely, the calculation of the similarity score and the compar-

ison of the certificate hashes. The latter step is implemented similarly to how it is done in more recent versions of the Android OS. The similarity score (the FSquaDRA similarity, or the fss score for short) corresponds to the Jaccard similarity coefficient (expressed by Formula 3.1) computed over the sets of file hashes extracted in Line 8. Line 23 prints the result of the comparison into a specified location (in our case, we print into a user-defined file).

$$jSim(H_k, H_l) = \frac{|H_k \cap H_l|}{|H_k \cup H_l|} \tag{3.1}$$

We implemented our algorithm in Java. We did not parallelize it intentionally (i.e., our tool runs in a single-thread program). This allows us to calculate the net time required to run our comparisons and predict the execution time and memory consumption. An increase of a dataset results in the linear growth of the execution time for attributes extraction, while the pairwise comparison operation cumulative time rises quadratically (in the number of apks under consideration). In the current implementation the memory consumption grows linearly with the number of applications.

3.3 Dataset description

Our dataset consists of 55779 Android applications. The dataset collection was performed during June-July of 2013. During this period we explored 8 different markets: the official *Google Play* [22] market and 7 third-party stores: *AndroidBest* [7], *AndroidDrawer* [8], *AndroidLife* [9], *Anruan* [10], *AppsApk* [11], *PandaApp* [32], and *SlideME* [35]. Table 3.1 lists all markets used in the analysis.

Table 3.1: Markets

Market	**Market link**	**# of apps**
AndroidBest	`http://androidbest.ru/`	1662
AndroidDrawer	`http://www.androiddrawer.com/`	2857
AndroidLife	`http://androidlife.ru/`	1678
Anruan	`http://www.anruan.com/`	4232
AppsApk	`http://www.appsapk.com/`	2679
Google Play	`https://play.google.com/store/apps`	13223
PandaApp	`http://android.pandaapp.com/`	14143
SlideME	`http://slideme.org/`	15305
Total		**55779**

During the analysis of alternative markets it was discovered that almost all of them, unlike the official one, provide a possibility to download applications without authentication. Therefore, to download an app it was simply required to get the right URL. Exploring the store webpages, we found out that for some markets such URLs can be easily predicted. For instance, all applications from the *AndroidBest* market can be downloaded using URLs tailing `download.php?id=n`, where n varies from 0 to the order number of the last uploaded application. In the case when the described method could not be applied,

e.g., for the *AndroidDrawer* market, a crawler was developed that parses `html` pages and extracts the URLs to download the applications.

To collect applications from the official Google Play store we developed a tool built on top of Google Play unofficial Python API[1]. Using this crawler we downloaded around 500 most popular free applications from each category.

After we downloaded the applications we performed a two-step cleaning of our dataset, filtering out the files that could not be recognized as valid Android applications and duplicates (those with same digests from the same market). The third column in Table 3.1 shows the final number of applications within each store. Our whole dataset occupies 317.4 GB of disk space.

3.4 Evaluation

We have run FSquaDRA on the collected app dataset on a Mac Book Pro laptop with 2.9 GHz Intel Core i7 Processor with 2 cores, and 8GB 1600 Mhz DDR3 memory. FSquaDRA required 15.10 hours to load all apk attributes in memory for our complete dataset, and 64.41 hours to compute the similarity scores for all apk pairs ($>10^9$) in our dataset consuming less than 6GB of RAM. On the dataset FSquaDRA performs on average 6700 app pair comparisons per second. We consider these results quite encouraging, as pairwise app comparison for code-based similarity metrics cannot be executed in comparable time.

Figure 3.2 presents a histogram of positive fss scores distribution for our dataset of 55779 applications (in logarithmic scale). Notice that the app pairs with $fss>0$ constitute approximately 5.41% of the total app pairs number for our dataset. To simplify presentation we break down the fss values into 10 bins in the range (0, 1]. In Figure 3.2 we can see that the vast majority of the application pairs with detected resource similarity have the fss score in the range (0, 0.1], and that for the fss score in the range (0.7, 1] there are more app pairs with the same certificate detected by FSquaDRA than app pairs with different certificates. We provide more insight why this is the case in the sequel.

To evaluate the quality of our approach and its ability to detect repackaging we would like to compare our results with some state-of-art code similarity-based repackaging detection technique, like [64, 88, 153, 154]. Unfortunately the authors have not released their code publicly, and we were not able to obtain it. Similar problem was also reported in [96], where the authors have resorted to use AndroGuard as a freely available tool for comparison of code similarity in apks. Following this approach, we use AndroGuard to provide us a metrics of code similarity for app pairs.

The main question we would like to investigate is whether the FSquaDRA similarity metrics is correlated with the AndroGuard code similarity metrics. This can be interpreted

[1] `https://github.com/egirault/googleplay-api`

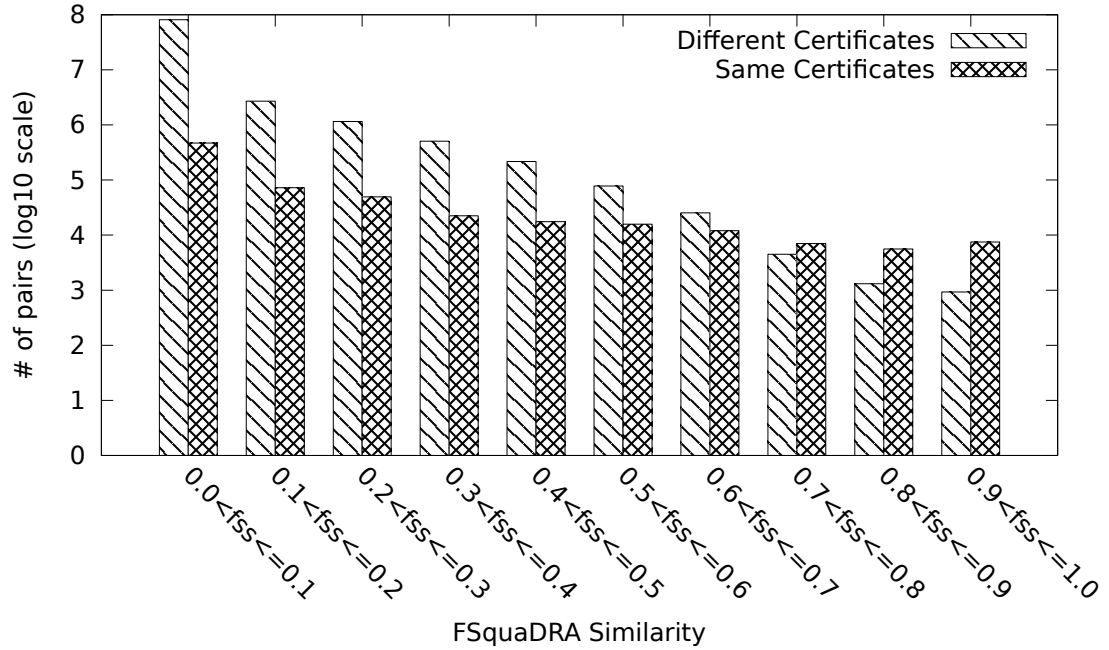


Figure 3.2: Histogram of app repackaging rates detected with FSquaDRA (logarithmic scale)

twofold:

- *False positives.* For applications that FSquaDRA classifies as similar, are they similar also according to the AndroGuard classification (and vice-versa)? If our tool classifies an app pair as similar, but there is no actual code similarity, this pair can be interpreted as false positive. It is obvious that it is impossible to completely avoid false positives for FSquaDRA because common resources, such as, e.g., open source sound and image files, can raise the FSquaDRA metrics, while the code would be different. So here we are interested in strong correlation of the similarity metrics values.

- *False negatives.* For applications that FSquaDRA classifies as completely different, are there many app pairs sharing code similarities according to AndroGuard? Again, it is not possible to completely avoid false negatives due to the different nature of code similarity and resource similarity, but we would like to assert that the false negatives rate is not too high.

Notice that in this section we interpret the AndroGuard code similarity score as ground truth. We have performed manual inspection of some application pairs to confirm the findings of FSquaDRA (reported further), but it is impossible to inspect manually substantial number of apks in our dataset. Therefore we have to rely on the code similarity metrics as the ground for evaluating FSquaDRA reliability.

Unfortunately, to compute a similarity value for two applications AndroGuard takes

significantly more time than FSquaDRA and it was not possible to compute the similarity metrics for the whole app corpus we have crawled. For instance, it takes approximately 65 seconds on average to compare one pair of apps using AndroGuard (the actual time of comparison depends a lot on the similarity of apps in the pair, it takes significantly less time to compare very similar apps than completely different ones). We cannot also rely on a straightforward random selection of app pairs, because it is clear from Figure 3.2 that, e.g., the share of app pairs with fss similarity in the range (0, 0.2] is a lot larger than the share of app pairs in (0.8, 1.0], which is as interesting. Therefore, we have performed a random selection of 100 app pairs with same certificate and 100 app pairs with different certificates from each bin with non-null fss metrics, and we have computed the AndroGuard similarity metrics (ags for short) for these pairs (2000 pairs total). This selection prohibits comparing the distributions of fss and ags values, but it enables the best selection of an app pairs corpus with different fss metrics, and without strong predominance of some fss value range.

To evaluate the false negative rates we have randomly selected 100 apk pairs with same certificate and 100 apk pairs with different certificates from the dataset with fss=0.

AndroGuard however was found to be not very reliable, as its similarity metrics was discovered to be not symmetric. That is, for two apks A and B, it could be that $ags^*(A, B) \neq ags^*(B, A)$, where ags^* is the value computed by the AndroGuard tool directly. We have decided to still use the existing AndroGuard implementation, but to adjust the AndroGuard score. We have experimented with a series of app pairs, and have established that the metrics $ags = (ags^*(A, B) + ags^*(B, A))/2$ is more faithful than the original ags^* similarity score, and we have used this metrics for comparison with FSquaDRA results.

Table 3.2 presents summary statistics computed for the randomly selected app pairs. Notice that for non-null fss values we compare separately app pairs with same certificate and with the different ones, as these two groups are different by nature. This observation is indeed reinforced by the data we have. Figure 3.3 presents and a scatterplot of the fss and ags similarity metrics values for the selected app pairs with different certificates (potentially plagiarised). We can see the strong correlation of the values from the figure. This is confirmed by the data: the standard Pearson's product-moment correlation computed for data in this figure is 0.791. Notice that any value $\geq$0.5 is commonly considered as strong correlation. Testing for the null-hypothesis (that true correlation is non existent) for this dataset gives that the 95% confidence interval is [0.767, 0.813]; and the p-value$\approx 10^{-16}$, so we can safely reject the null-hypothesis. The sample mean of the difference (fss-ags) for each selected app pair with different certificates is approximately equal to -0.047, with standard t-test rejecting the null-hypothesis (the p-value$\approx 10^{-12}$), and the 95% confidence interval for true mean [-0.052, -0.029]. The standard deviation for the difference (fss-ags) is 0.186. We also present a boxplot for this difference in Figure 3.4.

These data confirm that FSquaDRA can be an effective tool to detect repackaged

Table 3.2: Summary statistics for comparison of AndroGuard and FSquaDRA similarity metrics

Sample description	**Statistics name**	**Statistics value**	**Details**
App pairs with non-null fss with different certificates in comparison with ags; 1000 app pairs	Mean of difference fss - ags	-0.04122781	Standard one sample t-test 95% confidence interval: [-0.05278174, -0.02967388] p-value = 4.62e-12
	Standard deviation for difference fss - ags	0.1861895	
	Median	-0.04799	
	Correlation coefficient of fss and ags values	0.7919082	Pearson's product-moment correlation 95% confidence interval [0.7675988, 0.8139426] p-value ¡ 2.2e-16
App pairs with non-null fss with same certificates in comparison with ags; 1000 app pairs	Mean of difference fss - ags	-0.276119	Standard one sample t-test 95% confidence interval: [-0.2928976, -0.2593405] p-value = 2.2e-16
	Standard deviation for difference fss - ags	0.2703832	
	Median	-0.25180	
	Correlation coefficient of fss and ags values	0.580733	Pearson's product-moment correlation 95% confidence interval [0.5381128, 0.6203911] p-value ¡ 2.2e-16
App pairs with null fss with mixed certificates in comparison with ags; 200 app pairs	Mean of difference fss - ags	-0.04124	Standard one sample t-test 95% confidence interval: [-0.05152188, -0.03095351] p-value = 1.777e-13
	Standard deviation for difference fss - ags	0.07375432	
	Median	-0.01304	

applications, as the fss similarity values for app pairs with different certificates are highly correlated with code-based similarity metrics of AndroGuard; and the average difference in the similarity metrics produced by FSquaDRA and by AndroGuard is not significant.

Figure 3.5 presents a scatterplot of the fss and ags similarity metrics for the randomly selected apk pairs signed with the same certificate (potentially rebranded). The standard Pearson's product-moment correlation for this dataset is approximately 0.58 (the null-hypothesis on correlation is rejected, with 95% confidence interval for correlation [0.538, 0.62] , and p-value $\approx 10^{-16}$). This can be still interpreted as a strong correlation, but it is less strong than for the apk pairs with different certificate. The sample mean for the difference (fss-ags) in this dataset is approximately equal to -0.27 (standard t-test reports 95% confidence interval for true mean [-0.292, -0.259], and the null-hypothesis for sample difference mean being zero is rejected with p-value$\approx 10^{-16}$). This means that on average for apks signed with the same certificate FSquaDRA tends to estimate their similarity score noticeably lower than the code-based similarity score computed by AndroGuard. These findings can be intuitively explained by the fact that developers tend to reuse the code patterns across their products. For app pairs signed with the same certificate it is clear that they can contain similar code snippets with high probability. Therefore higher code similarity score is expectable.

We can also see from Figure 3.5 that there is a lot of app pairs with very high AndroGuard similarity score, but varying FSquaDRA similarity score, which are most probably the pairs impacting the correlation coefficient for this dataset. We have manually inspected some of these pairs and have managed to find several patterns, when such situations occur. One of the most common observed case is when the same code is used for

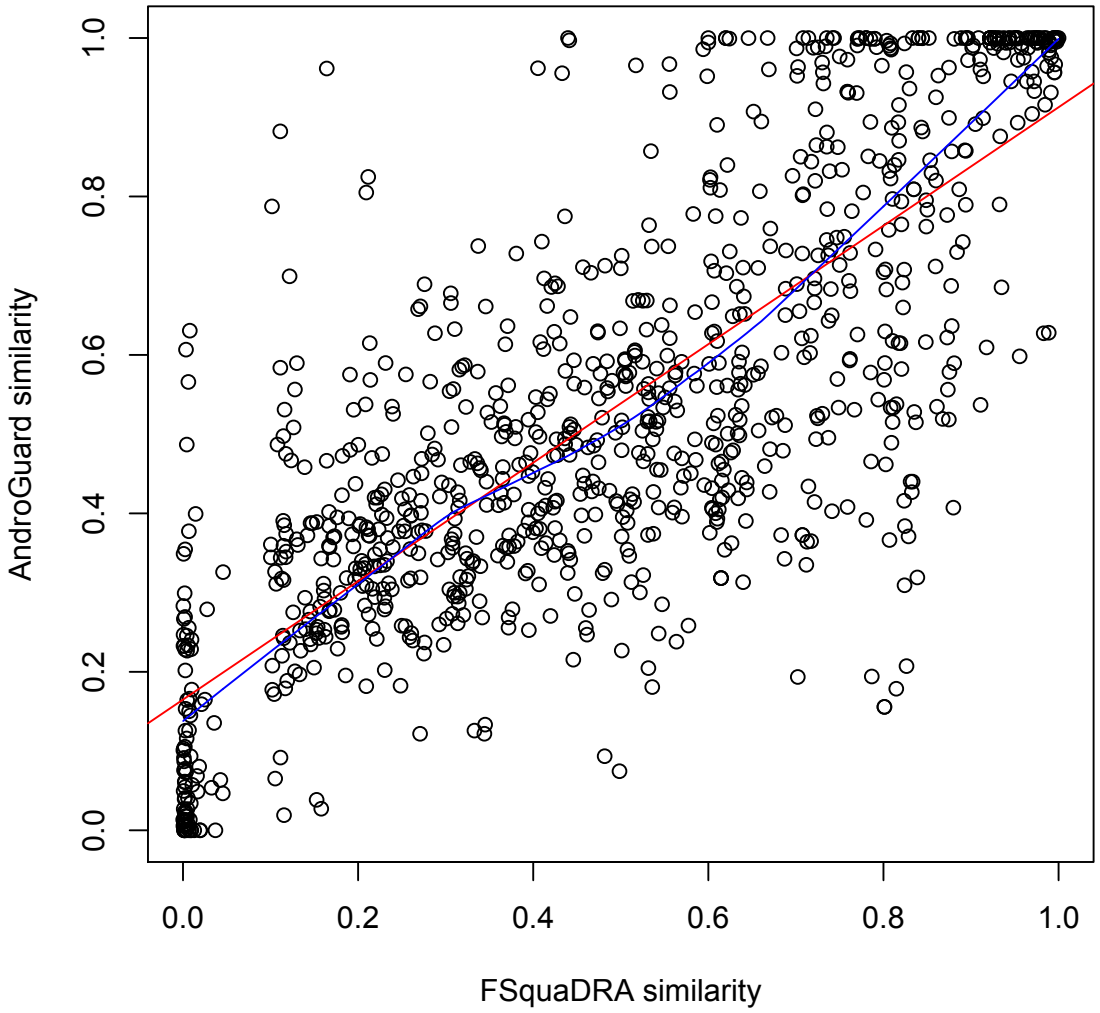


Figure 3.3: Scatterplot of FSquaDRA similarity between app pairs versus AndroGuard similarity for pairs signed with different certificates; the red line is the line of best fit, the blue curve is the LOWESS (locally weighted scatterplot smoothing line)

displaying different content. For instance, in our dataset we found several applications, which were developed to display books. For every book a single application has been developed. All these applications use the same code but the resources (the book chapters) are different. Thus, our tool shows low similarity score (because still some files, e.g. `classes.dex`, are the same), while according to the code similarity score the applications in the pair are the same. Similar behaviour we also witnessed with other categories of applications, which display the same type of content, e.g., for wallpaper apps and widgets. Another interesting example, which falls into this category, is when the apps in the pair provide a UI customization functionality for the third application. In this case, AndroGuard produces high similarity score for such pairs of apps, while because of the difference of the UI components FSquaDRA reports low similarity.

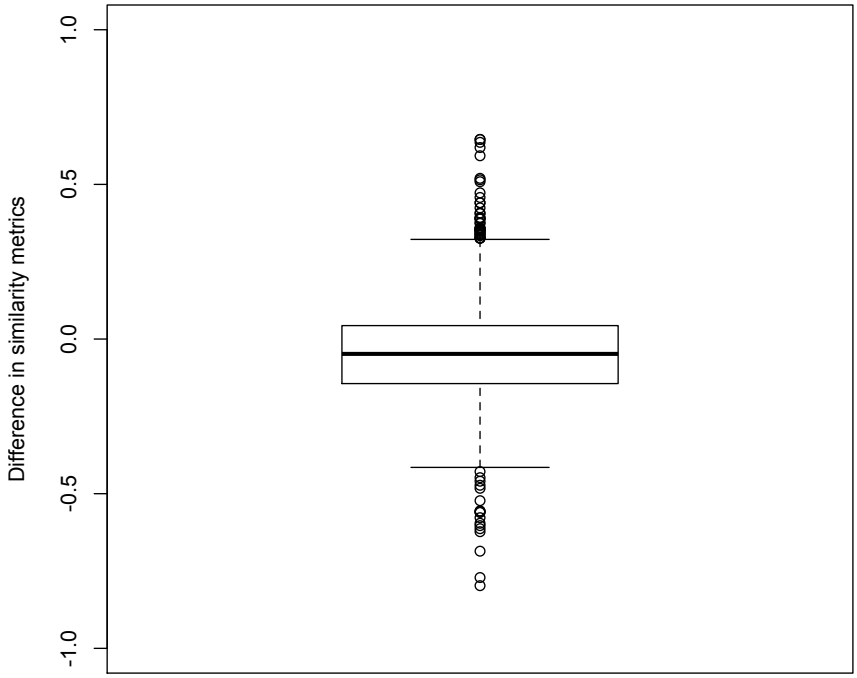


Figure 3.4: Boxplot of the difference of FSquaDRA similarity between app pairs and AndroGuard similarity for pairs signed with different certificates; for app pairs with fss>0

The lower correlation of the metrics can be also attributed to the usage of the same ad libraries. This happens when the fraction of the code produced by a developer significantly smaller than the ones brought by ad libraries. In this case AndroGuard falsely detects applications as repackaged, while FSquaDRA produces more credible results (because the applications are different).

Figure 3.6 presents a boxplot for the sample difference (fss-ags). In comparison with Figure 3.4, we can notice that for apk pairs with the same signature the range of the similarity scores difference is larger. Our data suggests that FSquaDRA may not be as efficient for detecting repackaging in apps signed with the same certificate (rebranded), as it is for the apps signed with different certificates (plagiarized). Nevertheless, correlation of the FSquaDRA score with the code-based similarity score of AndroGuard is still strong (>0.5).

Finally, Figure 3.7 presents a boxplot of the difference (fss-ags) for the randomly selected app pairs with fss=0 (100 apk pairs with the same certificate and 100 apk pairs with different certificates). Notice that on average FSquaDRA does not error a lot. Indeed, the sample mean of (fss-ags), or, simply, of the ags similarity score taken with the negative sign, is approximately -0.041, with the 95% confidence interval for the true mean [-0.051, -0.0309], and the standard deviation for this dataset is approximately equal to 0.0737. From these statistics and the boxplot we can see that for apk pairs not marked as similar by FSquaDRA AndroGuard does not see significant code similarity either,

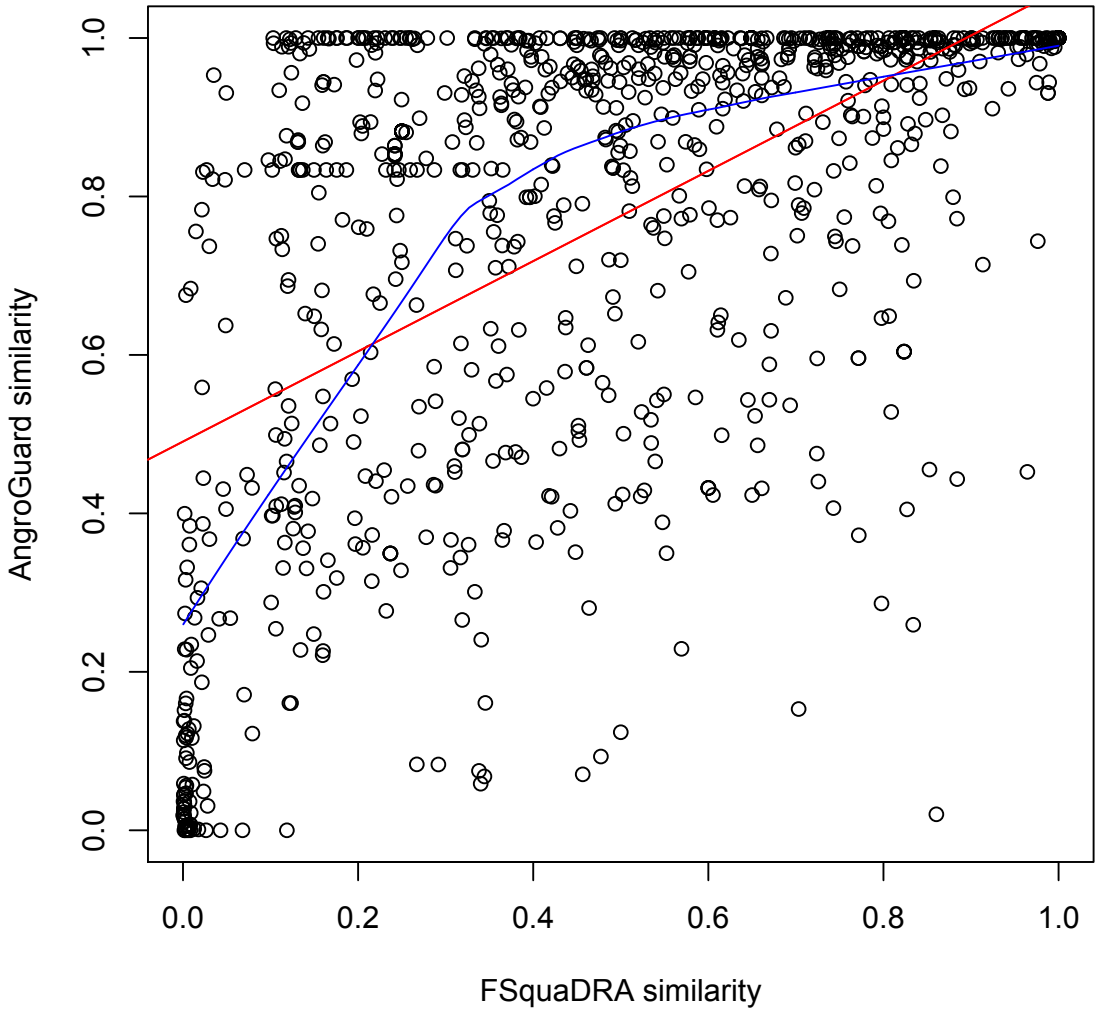


Figure 3.5: Scatterplot of FSquaDRA similarity between app pairs versus AndroGuard similarity for pairs signed with same certificate; the red line is the line of best fit, the blue curve is the LOWESS (locally weighted scatterplot smoothing line).

even for applications signed with the same certificate. Therefore we can conclude that if developers do not include any similar resouces in apps, they also mostly do not reuse code (this is often the case of apps produced by companies). We do not report the correlation coefficient for this type of dataset, as the fss score equals to 0.

Table 3.3 presents the summary cumulative statistics for the the randomly selected app pairs on which we have compared FSquaDRA with AndroGuard, and Figure 3.8 presents the boxplot for the sample mean of the difference (fss-ags) on these 2200 app pairs. Notice the very strong positive correlation coefficient (0.7149) for the values of fss and ags similarity scores on this dataset.

This data confirms that for plagiarized applications FSquaDRA can be an effective tool to detect repackaging, as the fss similarity metrics values for app pairs with different

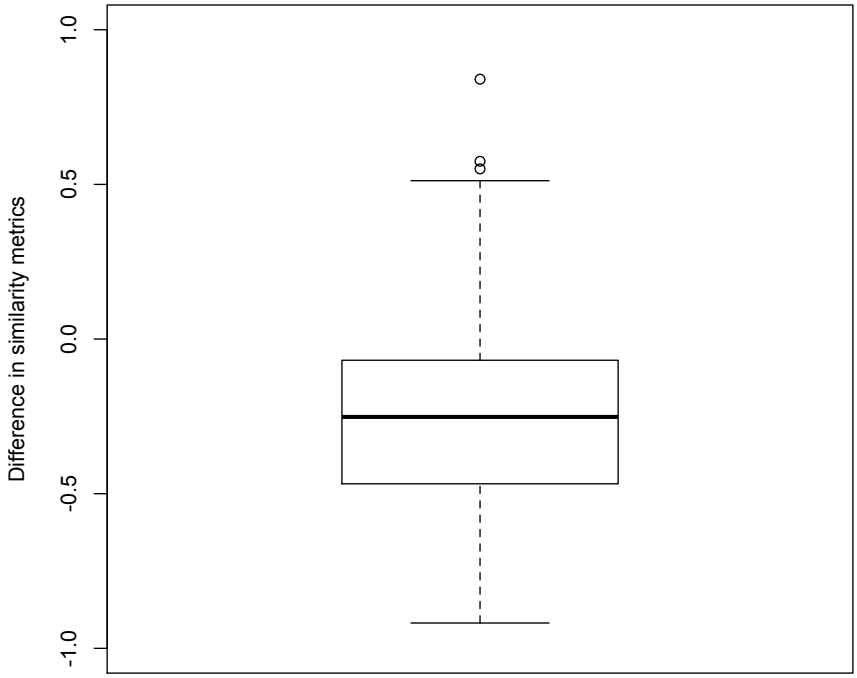


Figure 3.6: Boxplot of the difference of FSquaDRA similarity between app pairs and AndroGuard similarity for pairs signed with the same certificate; for app pairs with fss>0.

certificates are highly correlated with code similarity-based metrics of AndroGuard; and the difference in the similarity metrics produced by FSquaDRA and by AndroGuard is not significant.

3.5 Cross-Market Repackaging

After asserting that FSquaDRA produces similarity metrics that is valuable for detecting repackaged applications, being strongly correlated with the code similarity metrics, we look into repackaging rates corresponding to the markets under consideration, and investigate clusters of repackaged applications. Notice that clearly any FSquaDRA score greater than 0 for a pair of apks can be an indication that these apks are clones. However, to increase the certainty of detecting clones we have chosen the fss value of 0.7 to be a reliable threshold for repackaging. Based on our observations, we consider it a good starting point for resource similarity score sufficient to reliably detect clones, and we leave the task of identifying the threshold precisely for future work.

3.5.1 Cross-market Comparison

Table 3.4 presents the repackaging rates of Google Play applications cloned in other markets. Under the assumption that the Google Play market is the source of original

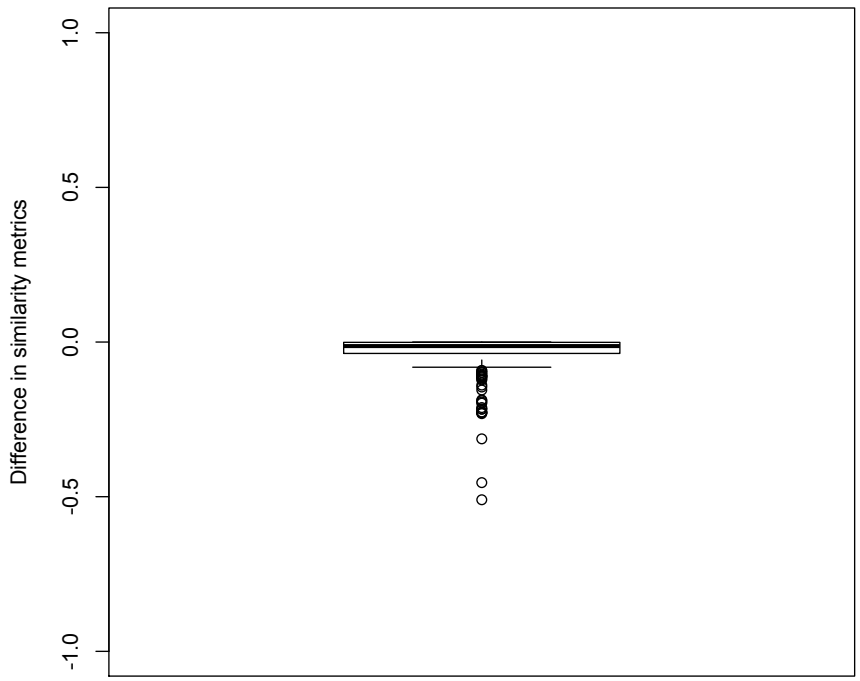


Figure 3.7: Boxplot of the difference of FSquaDRA similarity between app pairs and AndroGuard distance for app pairs signed with same and different certificates; for app pairs with fss=0.

applications, this table reports how many cloned pairs were detected with the fss score greater than 0.7, and the total number of apk pairs with fss>0 for all markets of our study compared with Google Play (the corresponding subset of our dataset). In this experiment we have compared each crawled apk in Google Play with each apk crawled in the considered third party markets. We also provide the processing time required for each market comparison with Google Play (Table 3.1 contains the number of apps crawled in each market). Notice that for all markets the number of app pairs with the fss score greater than 0.7 is not very significant. To understand better how the big is the subset of potentially repackaged applications we also provide the total number of app pairs with fss>0 detected, and the number of pairs with fss>0 and signed with different certificates.

From Table 3.4 we can observe that the markets with the highest repackaging rates are AndroidDrawer (16.16% of app pairs have similarity of resources fss>0) and Google Play (10.31% of app pairs have fss>0). We suspect that this is the case because these markets are more popular sources of apps, in comparison with others; and malicious repackagers that seek acquiring significant ad revenues or big user base for their botnets may target more popular markets. Yet, this intuition needs to be confirmed with more data, and there can be other plausible explanations.

Table 3.3: Summary statistics for comparison of AndroGuard and FSquaDRA similarity metrics for 2200 randomly selected app pairs

Sample description	**Statistics name**	**Statistics value**	**Details**
2200 app pairs, fss including app pairs with the same ags; and different certificates, and with fss=0 and fss>0	Mean of difference fss - ags	-0.14800	Standard one sample t-test 95% confidence interval: [-0.1585031, -0.1374917] p-value = 2.2e-16
	Standard deviation for difference fss - ags	0.2512748	
	Median	-0.09894	
	1^{st} quartile	-0.27380	
	3^{rd} quartile	0.00000	
	Correlation coefficient of fss and ags values	**0.7149053**	Pearson's product-moment correlation 95% confidence interval [0.6938442, 0.7347445] 99% confidence interval [0.6869681, 0.7407324] p-value<2.2e-16

Table 3.4: Results of experiments, each market in comparison with Google Play

Market	**Repackaging Rates**				**Time**	
	Same signature # pairs (fss>0.7)	Different signature # pairs (fss>0.7)	Total fss>0 (% of total pair #)	Total fss>0 with diff. cert. (% of total pair #)	Loading apk attributes in memory	Processing
AndroidBest	27	10	714258 (3.25%)	713194 (3.24%)	14.16 min	12.274 min
AndroidDrawer	528	14	6108547 (16.16%)	6097437 (16.14%)	15.46 min	56.02 min
AndroidLife	41	44	1145396 (5.16%)	1143400 (5.15%)	14.24 min	15.67 min
Anruan	106	97	3349271(5.985%)	3347895 (5.982%)	15.26 min	36.11 min
AppsApk	422	86	2105334 (5.94%)	2094716 (5.91%)	15.66 min	22.52 min
Google Play	1897	1301	9019858 (10.31%)	8985401 (10.27%)	13.28 min	59.97 min
PandaApp	755	381	10741872 (5.74%)	10726743 (5.73%)	28.52 min	136.65 min
SlideME	475	579	9496874 (4.69%)	9481029 (4.68%)	25.96 min	97.07 min

3.5.2 Application Clusters

Repackaged applications can form clusters (a set of repackaged apps stemming from some original application). We tried to elicit and analyze strongly connected clusters containing applications with very similar resources. The results produced by FSquaDRA can be interpreted as an undirected labelled graph, where nodes correspond to the applications in our dataset and edges represent similarity relationship between two applications, labelled with the fss similarity score. Thus, to find the clusters of applications we used the following algorithm. At first, we selected all pairs, which had shown the FSquaDRA similarity value more than 0.7. After that in the resulting graph we searched for connected components (i.e., set of connected nodes), which corresponded to application clusters. We looked for clusters that have 3 and more nodes. Using this approach we discovered 71 cluster, the largest of which included 9 applications.

We have investigated manually some of the clusters, and we report on the largest two of them (smaller clusters are not reported for the lack of space). The largest cluster with 9 nodes contains applications from 3 different markets (4 from Google Play, 4 from SlideME and 1 from AppsApk), all signed with different certificates. The nodes are connected with 8 edges; similarity scores for app pairs not connected by an edge vary in the range [0.61, 0.7). The cluster with 8 applications contains packages distributed on 5 different markets (2 come from Google Play, 3 from SlideME, 1 from Anruan, and 2 from PandApp). These

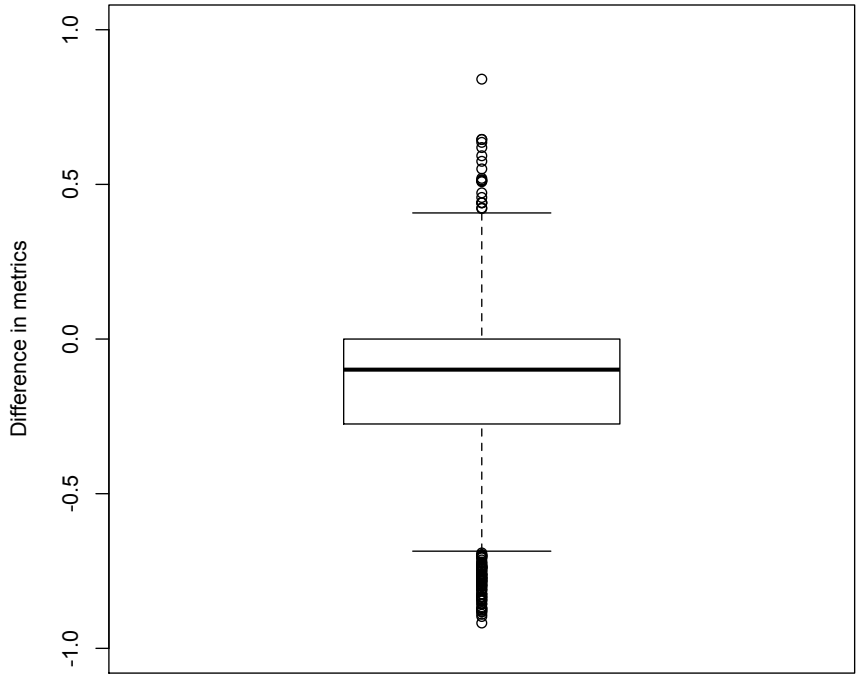


Figure 3.8: Boxplot of the difference in the FSquaDRA similarity and the AndroGuard similarity for all randomly selected pairs (2200 pairs)

8 applications are connected by 7 nodes, and the fss scores for the app pairs not connected by an edge vary in [0.4, 0.6). In this cluster 3 applications (from Anruan and PandApp) were signed by the same certificate, and others were signed with different certificates.

After we manually inspected all applications in these clusters, we discovered that these apps were legitimate applications and not maliciously repackaged. These "false positives" appeared because all apps in the cluster used the same popular library ActionBarSherlock [1], which is supplied with lots of files. Additionally, the applications contained a very limited number of their own unique files, and thus FSquaDRA falsely detected them as repackaged applications. We performed also an analysis using AndroGuard and found out that the code files were also very poisoned with this library. AndroGuard similarity scores for these clusters were in the range [0.46, 0.96]. Therefore, in the shadow of the methodology selected for our analysis this is still a good result for our tool. However, this example clearly shows that it is desirable to implement techniques for automatic library resources detection and exclusion, similarly as it is done for code in [64, 153]. We leave this problem for the future work.

3.6 Related work

The AndroGuard algorithm which computes the similarity score of two applications is presented in [66]. The similarity score is based on the analysis of Dalvik code of an app pair

and detection of identical, similar and different (new or deleted) methods in the apps. To perform this, the algorithm a) generates a signature for each method of each application, b) identifies all methods that are identical in both apps, c) discovers all methods that are similar. A signature is generated based on the method control flow information, used API calls and exceptions inside the method. If two signature hashes are identical then the methods are considered identical. To compute the similarity between methods Normalized Compression Distance (NCD) [60] is used. The final similarity score is computed taking into account similarity of methods. This algorithm can be used to analyze if an application has been plagiarized comparing it with the original. The similarity score produced by the algorithm is a clear indicator to a developer if her application was repackaged.

In [154] the authors analyze repackaged applications in third-party markets in comparison with Google Play. To perform this analysis the authors have developed a tool called DroidMOSS, which compares a pair of applications and produces a similarity score. If similarity score is higher than 70% the applications are considered repackaged. The algorithm uses fuzzy hashing of app code to produce a short fingerprint and then compute the edit distance between these two fingerprints. The authors crawled 22,906 apps from 6 third-party markets and 68,187 applications from Google Play. Then they selected 200 apps from each market and compared them pairwise with all applications from the official market. Their analysis shows that 5-13% applications hosted in alternative markets are repackeged. These conclusions agree with our findindings reported in Section 3.5.

In the paper [153] the authors further investigate the problem of repackaged applications and concentrate on detection of piggybacked applications. Piggybacked apps are repackaged applications that carry a payload in addition to main functionality. The authors observe that usually the payload is loosely coupled with the rest code of the application. Thus, to find the piggybacked applications the authors perform module decoupling to primary and non-primary modules, clustering nodes in program dependency graph. After that for each primary module a fingerprint is computed, which is used to distinguish primary module functionality. After that while iterating over the fingerprints the linearithmic algorithm detects the applications with similar primary modules, which are considered as piggybacked candidates. Finally, comparing the sets of non-primary modules of these similar applications, piggybacked applications are detected. The experiments reported by the authors show 1.3% of piggybacked apps in the considered dataset. False negavive rate of the PiggyApp tool is reported as 5.2% One the most important feature of PiggyApp that it can find repackaged applications not considering the signature of applications.

Paper [63] presents a tool called DNADroid to detect cloned (plagiarized) applications. Using the semantic similarity of applications the tool detects potential clone candidates. At the second step, the tool extracts Program Flow Graph (PDG) of each method of compared applications, applies the *lossless* and *lossy* filters, which remove PDGs of some

methods from consideration and, based on the *subgraph isomorphism* as a final criteria of method similarity, computes similarity score of the applications. To assess DNADroid, the authors collected 75,000 applications from 13 different third-party markets. After the first stage of detecting potential clones they randomly selected 9,400 pairs and applied the second step. The evaluation shows that on average the speed of processing of applications on the second stage is 0.71 app pair/minute. Having 70% threshold, from these pairs DNADroid managed to detect 191 pairs which are cloned. Manual analysis showed that DNADroid produces 0% of false positives. The authors also compared their tool with AndroGuard [2]. On 191 pairs AndroGuard failed for 24 pairs and produced very low similarity score for 10 pairs meaning that it missed 18% of the pairs found by DNADroid. We would like very much to use DNADroid, or its successor AnDarwin in our study, but unfortunately it is not publicly available.

Continuing the work on DNADroid [63] Crussell et al. developed a new tool AnDarwin [64], which unlike DNADroid does not compare applications pair-wise but instead extract features from app code and compares them. This allows the developers to perform large-scale analysis of Android applications. AnDarwin is used to find plagiarized and rebranded apps. Using a dataset of 265,359 third-party applciations collected from 17 different markets they managed to detect 4,295 cloned applications and 36,106 rebranded applications. Moreover, the authors showed the ability of this tool to detect new samples of known malware families. The tool operates in the following way. At first, it represents each app as a set of *semantic vectors* using information extracted from PDF of each method. Then it detects similar parts of code clustering the vectors obtained previously. Code that appears in many apps is eliminated from the consideration because it represents different libraries used during the development of Android applications. Finally, using partial and full similarity of semantic vectors it detects similar applications. The analysis showed that AnDarwin's false positive rate is about 3.72%. We plan to rely on the approach of AnDarwin for eliminating libraries in the future. The authors also performed in [85] a large-scale analysis of impact of cloned applications. By using mobile traffic data from a 1-tier US cellular carrier and the collected app dataset, they assessed the impact of Android application plagiarism by analysing the proportion of revenues from advertisement clicks collected by plagiarized applications and the original ones. To eliminate the effect from rebranded applications, the authors before the analysis merged the same developers (based on the certificate used to sign an app and based on the client ID used to uniquely identify the recipient of money for advertisement). The analysis shows that the developers of original apps loose about 14% of advertisement revenues and about 10% of user base due to application plagiarism.

The authors in the work [118] concentrate on investigation which applications are likely to suffer from being plagiarised, and how to detect plagiarised applications uploaded to a market. The authors analysed the meta-information of 158,000 applications.

They detected that 29.4% of applciations are more likely to be plagiarised, based on the assumption that it was more likely that a malicious developer wouldl use for plagiarising the applications, which alredy contained the permissions needed to perform malicios actions. To detect plagiarized applications the authors proposed three schemes: Symbol-Coverage, AST-Distance and AST-Coverage, which are based on symbol tables and Abstract Syntactic Tree fingerprints. To assess the effectiveness of the developed schemes the authors randomly selected the 7,600 applications and built Pseudo-Store. Then they obtained a set of 15 pairs of original and plagiarised applications (which contained embedded HongTouTou malware) and put them into Pseudo-Market. The results show that AST-Coverage scheme successfully detects the upload of plagiarised application into Pseudo-Market. The reported false-positive rate consitutes 0.5%. Moreover, the authors show that the proposed method is resilient to different obfuscation techniques.

In the paper [137] the authors concentrated on the analysis of third-party markets. In particular, they analysed 195 third-party markets, including Google Play, crawling totally 76,480 applications. They also selects two classes of plagiarised applications: spoofing and grafting. Spoofing apps present little or none of the functionality of the original application, while grafting applications consist of an existing original app with embedded malware in it. To assess the number of malicious applications in different market the authors uploaded them to VirusTotal and analysed the results. They found that some markets completely consist of repackaged applications. To combat with such kind of markets the authors developed a tool called AppIntegrity. During the installation of an application this tool parses the package name of the app and extracts the domain name. Then it requests this domain for public key. The requested public key is used as an additional step to verify that the application has been developed by the developer who owns the domain name. It could be interesting to see in the future if FSquaDRA performs well on spoofing apps (we expect it to perform well on grafting apps).

The paper [88] presents another approach to detect code reuse among Android applications. To discover the similarities between the code they use k-grams of Dalvik opcode sequences as features. To obtain application representation they apply hashing to the extracted features. Thus, the app representation is efficient, and they can compute pairwise application similarity for large number of apps. The Juxtapp tool implementing this algorithm can detect (a) buggy and vulnerable code reuse (b) known malware instances and (c) pirated applications. To detect these kind of applications Juxtapp need as an input the code that it has to find (for instance, to detect pirated applications Juxtapp needs original applications). To assess the Juxtapp efficiency the authors ran experiment of pairwise comparison on a set of 95.000 Android applications. One of the experiments was run on Amazon EC2 cluster with 25 slave nodes, each having 8 virtual cores and 68.4Gb of memory. The experiment continued about 200 minutes, about 75 of which were spent on feature extraction. During the experiments from a dataset of 30.000 applications from

Android Market the authors identified 174 applications containing vulnerable patterns in the in-app billing code and 239 apps containing those in the code, which uses Licence Verification Library(LVL). Moreover, they identified presence of 34 new instances of known malware in the alternative Anzhi market. At the same time, according to the authors Juxtapp cannot detect spoofing applications.

Recently, a framework for evaluating Android application repackaging detection algorithms has been proposed [96]. In the paper the authors classify currently available approaches for detection of repackaged applications and presents a framework that can be used to assess the effectiveness of this kind of algorithms. The framework operation resembles the way how an adversary repackages an app now. At first, he converts Dalvik bytecode into an intermediate representation (IR), then changes something in the code (for instance, during cracking of an application he may remove the checks whether the app has been bought) and, finally, converts the obtained version back into Dalvik bytecode. Moreover, on the second step the adversary may use obfuscation tools that change the code and can prevent the detection of repackaging. Similarly, the framework prorposed in [96] translates Dalvik bytecode into Java code, applies obfuscation techniques and packs back the code into Dalvik representation. During the analysis the authors use the SandMark tool [61], which provides a wide range of obfuscation techniques. The authors proposed to assess repackaging detection algorithms by *broadness* (i.e., how an algorithm can stand to obfuscation techniques applied separately) and by *depth* (i.e., if an algorithm is resilient to techniques applied sequentially). As the case study, the authors applied the framework to AndroGuard [2] – the only publicly available tool for repackaging detection. The results show that AndroGuard can successfully combat with different obfuscation techniques and, thus, can be widely used to detect repackaged applications.

Chapter 4

Static-Dynamic Analyser of Android Apps

Static analysis of Android applications can be hindered by the presence of the popular dynamic code update techniques inherited from Java: dynamic class loading and reflection. For example, recent Android malware samples specifically use dynamic code updates to conceal their malicious behavior from static analysis. These techniques defuse even the most recent static analyzers (e.g., [68, 152]) which explicitly make a closed world assumption. We propose an approach that augment the information available to static analyzers. It combines static and dynamic analysis of Android applications in order to reveal the hidden/updated behaviors and extends the method call graph of the application under analysis with this information. We also report the results of evaluation of our implementation of the tool called StaDynA.

The rest of this chapter is organized as follows. Section 4.1 introduces the problem of dynamic code updates in Android applications. Section 4.2 presents the results of our study of dynamic code update techniques in benign and malicious apps. Section 4.3 motivates our research presenting a real case of dynamic code updates usage to conceal malicious behavior. Section 4.4 presents our framework. Sections 4.5 and 4.6 provides a background on dynamic class loading and reflection on Android. Section 4.7 details the StaDynA implementation. Section 4.8 presents our approach to build method call graphs and visualise them. Section 4.9 details the evaluation of StaDynA on real apps. Section 4.10 discusses related work.

4.1 The Problem of Dynamic Code Updates

Mobile applications (apps for short) are complex programs that offer sophisticated user experiences by exploiting the whole spectrum of dynamic features offered by modern programming languages such as Java.

Yet, these very features of the language (e.g., Java's reflection) combined with the

common practices adopted by mobile app developers (e.g., dynamic code updates) make the static analysis of mobile apps a challenging task. This is particularly daunting when static analysis is used in order to check the security of the mobile application (e.g., to detect the presence of malicious behavior). Indeed, Rastogi et al. [121] mention reflection among the techniques that render most of the current static analysis tools unable to detect malicious code.

Specifically, static analysis is hindered by code that can evolve dynamically, because some paths in the code are impossible to discover or to traverse as they are created at runtime. As a matter of fact existing state-of-art statical analysers (e.g., [68, 152]) make the "closed world" assumption that the code base does not change dynamically. This is a clear simplification of what happens in the real world, where many popular, legitimate apps use dynamically loaded code. Previous approaches that enhanced static analysis of Java code in presence of dynamic code update techniques (e.g., [51]) cannot be directly applied to Android due to the differences in the platforms.

Even manual reviews do not seem to offer a solution as demonstrated by Wang et al. [138] who reported a proof-of-concept iOS app that passed successfully the App Review process by Apple because its submitted code was benign. Yet the app was able to dynamically update the code on the device in order to introduce malicious control flows and to perform illicit tasks (such as attacking other apps and exploiting kernel vulnerabilities).

In this chapter we provide a solution to the problems of static analysis in presence of dynamic code updates. Our approach complement and complete existing static analyzers, and is efficient in uncovering hidden dangerous behaviors in Android apps.

The results of StaDynA can then be fed back to app analyzers for a more refined analysis. It is a powerful approach that can enhance the performance of existing static analyzers and malware detectors.

4.2 Study of Dynamic Code Updates in Apps

To understand how significant is the use of reflection and dynamic class loading (DCL) in Android apps we performed a study of 13.863 packages obtained from Google Play [22] and 14.283 apps from several third-party markets gathered in July of 2013, along with 1260 malware samples [156]. Notice that for reflection we only consider calls which influence the method call graph, i.e., method invocation (`invoke`) and constructor calls (`newInstance`). All other reflection usage cases are left outside of this analysis.

4.2.1 Google Play

Google Play is the official market of the Android applications, which is supported by Google. We downloaded about 500 top free applications from each category from Google

Play. Table 4.1 contains the results of our analysis.

Table 4.1: Analysis of Google Play apps

Category	# App	DCL used by	# DCL calls	Refl. used by	# refl. calls
APP_WALLPAPER	518	120	124	464	6183
APP_WIDGETS	492	119	135	468	15651
BOOKS_AND_REFERENCE	510	107	107	452	7899
BUSINESS	504	38	42	429	8229
COMICS	517	94	98	439	5588
COMMUNICATION	502	85	98	398	9768
EDUCATION	510	118	127	475	9376
ENTERTAINMENT	507	134	136	481	10551
FINANCE	519	78	79	459	8599
GAME	520	202	237	512	19525
HEALTH_AND_FITNESS	520	77	82	469	11696
LIBRARIES_AND_DEMO	520	88	103	369	4188
LIFESTYLE	517	104	109	473	8933
MEDIA_AND_VIDEO	516	118	126	455	9864
MEDICAL	518	64	65	432	6153
MUSIC_AND_AUDIO	513	98	107	456	11365
NEWS_AND_MAGAZINES	501	79	83	464	11801
PERSONALIZATION	516	90	93	452	7988
PHOTOGRAPHY	519	149	168	488	14569
PRODUCTIVITY	516	77	89	454	12082
SHOPPING	519	50	53	470	11080
SOCIAL	520	119	122	489	15990
SPORTS	520	97	102	473	9283
TOOLS	520	90	108	459	9601
TRANSPORTATION	516	58	58	424	7113
TRAVEL_AND_LOCAL	496	50	50	421	9184
WEATHER	517	70	76	408	6331
Total:	**13863**	**2573**	**2777**	**12233**	**268590**

The analysis shows that on average 19% of all applications in Google Play contain dynamic class loading calls. The categories *"BUSINESS"* (8% of apps), *"SHOPPING"* (10%) and *"TRAVEL_AND_LOCAL"* (10%) have shown the minimum percent, while in the *"GAME"* category about 39% of the applications exploit dymanic class loading functionality. The results can be easily understandable. During the past several years the games for mobile platforms have evolved considerably. Almost all of them provide great user interface, original gameplay and realistic physics. All this requires from a developer to write tons of code for different versions of platform. Thus not surprisingly, the developers choose a strategy when an original application (which size is limited to 50 Mb) is only an installer, which downloads additional code from a server and loads it dynamically. On average, there is 1.08 calls of dynamic class loading functions per application.

Considering reflection, 88% of all applications use reflection calls that are of interest for our system. In case of the *"GAME"* category the percent reaches 98%, showing that almost each applciation in this category relies on reflection. Additionally, the applications from this category show also the highest average number of reflection calls (38.13), while on average there are 21.96 reflection calls per app.

4.2.2 Third-party markets

This subsection reports on the result obtained during the analysis of applications retrieved from various third-party markets. We downloaded applications from 6 third-party markets, namely, *AndroidBest* [7], *AndroidDrawer* [8], *AndroidLife* [9], *Anruan* [10], *AppsApk* [11] and *F-Droid* [18]. The first 5 markets contain only apk files, while the latter (*F-Droid*) distributes only open-source applications, which sources along with the final packages can be found in this market. For each market, the cleaning procedure has been performed, i.e., the files with the same digest of content considered as the same apps, thus, only one representative with the same hash has been left. Although some of these markets also divide the applications into categories, due to the peculiarities of the app crawling process this information has been lost. Thus, in this section we the values average across a market. The results of this analysis can be found in Table 4.2.

Table 4.2: Analysis of third-party market apps

Market	# App	DCL used by	# DCL calls	Refl. used by	# refl. calls
AndroidBest	1655	35	58	1088	11598
AndroidDrawer	2677	379	516	2596	85466
AndroidLife	1677	117	138	1368	24921
Anruan	4230	162	250	2868	43444
AppsApk	2664	112	159	1907	29473
F-Droid	1380	11	11	792	10775
Total:	**14283**	**816**	**1132**	**10619**	**205677**

It can be mentioned that while for Google Play the average number of applications, which uses dynamic class loading, are about 19%, in case of third-party markets this percentage constitutes only 6%. This can be explained by the fact that we downloaded all available applications from third-party markets, while in case of Google Play we did only top applications, which expose complicated functionality that requires loading code from external sources. Moreover, the *F-Droid* market also influence on this value because only 1% of its applications use dynamic class loading functionality. At the same time, the number of dynamic class loading calls is higher in third-party markets then in Google Play. It should be mentioned that the lowest percent of applications with dynamic class loading calls are observed for the *F-Droid* market, which contains only open-source applications. This shows that among the open-sources developers there is a lower aspiration of dynamic class loading usage.

As for the reflection, about 74% of the appls downloaded from third-party markets use it. Similarly as for the dynamic class loading case, this percent is lower than for *Google Play* applications. At the same time, the applciations from the *AndroidDrawer* market have very high percent (97%) of the reflection usage. The average number of reflection calls is 19.37 across all third-party markets, which fall downs to 13.6 calls for the *F-Droid* market.

4.2.3 Malware

Additionally to the analysis of benign applications, we perform a study of malware samples, provided by Zhou et al. [156]. The results of the analysis is reported in Table 4.3, where a row represents the numbers obtained for one malware family.

Table 4.3: Analysis of malware

Malware Family	# App	DCL used by	# DCL calls	Refl. used by	# refl. calls
ADRD	22	0	0	19	56
AnserverBot	187	187	890	187	1514
Asroot	8	0	0	0	0
BaseBridge	122	42	84	112	632
BeanBot	8	0	0	8	25
Bgserv	9	0	0	9	19
CoinPirate	1	0	0	1	4
CruseWin	2	0	0	0	0
DogWars	1	0	0	1	1
DroidCoupon	1	0	0	1	5
DroidDeluxe	1	0	0	1	2
DroidDreamLight	46	0	0	43	201
DroidDream	16	0	0	10	40
DroidKungFu1	34	0	0	7	65
DroidKungFu2	30	0	0	2	12
DroidKungFu3	309	2	4	294	1403
DroidKungFu4	96	6	8	87	518
DroidKungFuSapp	3	0	0	3	3
DroidKungFuUpdate	1	0	0	1	2
Endofday	1	0	0	1	12
FakeNetflix	1	0	0	0	0
FakePlayer	6	0	0	0	0
GamblerSMS	1	0	0	1	3
Geinimi	69	0	0	69	176
GGTracker	1	0	0	0	0
GingerMaster	4	0	0	4	8
GoldDream	47	3	6	39	127
Gone60	9	0	0	0	0
GPSSMSSpy	6	0	0	0	0
HippoSMS	4	0	0	3	3
Jifake	1	0	0	1	31
jSMSHider	16	0	0	16	80
KMin	52	0	0	41	41
LoveTrap	1	0	0	0	0
NickyBot	1	0	0	1	6
NickySpy	2	0	0	0	0
Pjapps	58	0	0	28	225
Plankton	11	11	11	11	25
RogueLemon	2	0	0	0	0
RogueSPPush	9	0	0	0	0
SMSReplicator	1	0	0	0	0
SndApps	10	0	0	0	0
Spitmo	1	0	0	0	0
Tapsnake	2	0	0	0	0
Walkinwat	1	0	0	0	0
YZHC	22	0	0	1	1
zHash	11	0	0	11	33
Zitmo	1	0	0	0	0
Zsone	12	0	0	12	36
Total:	**1260**	**251**	**1003**	**1025**	**5309**

The average percent of dynamic class loading usage across all malware samples shows the highest value (20%). At the same time, the samples of the *AnserverBot* family have made a major contribution to this percent (187 out of 251 applications, which use dynamic class loading, are represented by this family). To our knowledge, the low percent may be

explained by the fact that the dynamic class loading technique has become popular only recently, while the dataset mainly contains the samples collected in the previous years. However, it can mention that in two families (*AnserverBot* and *Plankton*) each sample contains dynamic class loading calls. This shows that the malware of these families rely on dymanic class loading functionality for their operation, which is confirmed by the reports [99, 155].

Considering the reflection usage in malware, it can be mentioned that about 81% of all samples use this technique. At the same time, not all the examples of reflection usage are caused by malicious functionality. According to [156], 86% malware samples are based on the repackaged applications. Thus, the reflection usage can be also exploited in the benign versions of the repackaged apps. This is also proved by the fact that there are families, where not all samples expose the usage of reflection, although the percent of such samples can be quite large.

4.3 Illustrative Example of Dynamic Code Update

Listing 4.1 is a code snippet from the *AnserverBot* trojan [155], which illustrates how reflection and DCL are used to hamper static analysis for malware detection. Line 18 shows an example of dynamic class loading in Android using the `DexClassLoader` class. The path to the file (stored in the `str3` variable), from which the code is loaded, is computed at runtime. It consists of two parts; the first points to the application data directory, while the second specifies the name of the file in an encrypted form (see Line 10). Line 18 also presents the Class object obtained using the method `loadClass`, which name is specified by the `paramString1` parameter.

Line 28 shows how to obtain an object of a class using the reflection call of the default constructor. Line 30 demonstrates a method invocation. The name of the invoked method is passed as a parameter (see Line 26) and, thus, may not be available for static analysis.

4.4 An Overview of StaDynA

Logically, the analysis process in our system can be divided into two phases: static analysis and dynamic analysis. At the same time, in the running system these two phases are closely interleaved providing each other required information. However, it is easier to consider the phases in a sequential order although some events may happen simultaneously in the working system. The architecture of StaDynA presented in Figure 4.1 comprises two logical components: a server and a client.

The server is responsible for the performing the static analysis of an application. It builds the initial *method call graph* (MCG) of the app, integrates the results of the dynamic analysis of the information coming from the client, and stores the results of the analysis.

```
[com.sec.android.providers.drm.Doctype]
public static Object b(File paramFile, String paramString1, String paramString2, Object[]
paramArrayOfObject)
{
  String str3;
  if (paramFile == null) {
    String str1 = a.getFilesDir().getAbsolutePath();
    //get the name of the file to be loaded
    //9CkOrC32uI327WBD7n__ -> /anserverb.db
    String str2 = Xmlns.d("9CkOrC32uI327WBD7n__");
    str3 = str1.concat(str2);
  }
  for (File localFile = new File(str3); ;localFile = paramFile) {
    String str4 = localFile.getAbsolutePath();
    String str5 = a.getFilesDir().getAbsolutePath();
    ClassLoader localClassLoader = a.getClassLoader().getParent();
    //get the class specified by "paramString1" from anserverb.db
    Class localClass = new DexClassLoader(str4, str5, null, localClassLoader).loadClass(
    paramString1);
    Class[] arrayOfClass = new Class[5];
    arrayOfClass[0] = Context.class;
    arrayOfClass[1] = Intent.class;
    arrayOfClass[2] = BroadcastReceiver.class;
    arrayOfClass[3] = FileDescriptor.class;
    arrayOfClass[4] = String.class;
    //get the method specified by "paramString2"
    Method localMethod = localClass.getMethod(paramString2, arrayOfClass);
    //create new instance of the class
    Object localObject = localClass.newInstance();
    //call the corresponding method with arguments in array "paramArrayOfObject"
    return localMethod.invoke(localObject, paramArrayOfObject);
  }
}
```

Listing 4.1: Illustrative example: DCL and reflection usage in `AnserverBot`

The client part is a modified Android operating system, hosted either on a real device or an emulator. The client runs the application whenever the dynamic analysis is required.

From a logical perspective, our system interleaves the execution of static and dynamic analysis phases. To simplify the presentation, we describe them sequentially.

Preliminary Analysis. The server statically analyzes an app package and builds a MCG of that application (see Step *a* in Figure 4.1; solid arcs denote edges resolved statically from the apk). Dynamically loaded code is not present in the MCG built during this phase. Further, a method called through reflection may also not be inferred if the name is represented as an encrypted string. Sometimes DCL and reflection are used together, the former loads new code into memory, whose parts are then called by using the latter. In Figure 4.1 the dashed arcs and a part of the MCG in the dashed oval represent the MCG parts that cannot be resolved statically, but can be discovered by StaDynA. Still, a static analyzer can effectively detect the points in the MCG where the functionality of an application may be extended at runtime. Indeed, the usage of reflection and DCL requires the usage of specific API calls provided by the Android platform. The server detects these calls during the static analysis phase by searching for methods, where DCL and reflection API calls are performed. We call these methods *methods of interest (MOI)*.

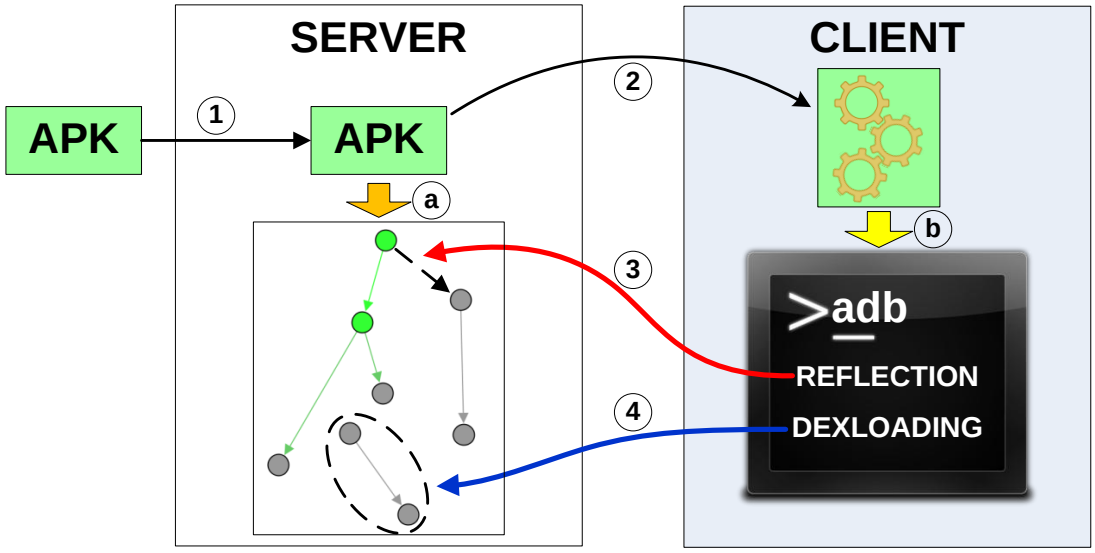


Figure 4.1: System overview

Dynamic Execution. If MOIs have been detected in the application, StaDynA installs the app on the client (Step *2*) and launches the dynamic analysis. The dynamic phase is exercised to complement the MCG of the app and to access the code loaded dynamically. The dynamic analysis is performed on a device (or an emulator) with a modified Android OS. The added modifications log all events when the app executes a call using reflection, or when additional code is loaded dynamically. Along with these events, the client also supplies some additional information. E.g., in the case of a reflection call, the information about the called function and the stack trace (it contains the ordered list of method calls, starting from the most recent ones) is added. Thus, the first function after reflection or DCL calls must correspond to a MOI detected during the static phase.

Analysis Consolidation. The information collected by the dynamic execution is passed back to the server side (Step *3*). The server analyzes it and complements the MCG of the app with a new edge (in Figure 4.1 it is represented by a dashed arc), which connects the node of the method that initiated the call through reflection (the node at the beginning) with the one corresponding to the called function (the node at the end).

When DCL is triggered the client infers which file was used to get the code from. Using this evidence the server downloads the file (Step *4*) with the code, and performs the static analysis on it. The MCG of the app is then updated with the obtained information. Additionally, for each downloaded file the server analyzes whether it contains other MOIs. If it does, the list of the MOIs for the application is updated. This allows StaDynA to unroll nested MOI. Similarly to the reflection case, the server also obtains the stack trace data that are used to detect the source of the DCL call.

The process is then re-iterated until the analyst is satisfied.

Marking Suspicious Behaviour. The dynamic capabilities (reflection and DCL) cannot help the malicious app to bypass the sandbox perimeter. At the same time, the dynamic behaviour of an app can conceal a malicious payload from the static and dynamic analysis of Google Bouncer [117]. Yet, the recent studies show that users simply agree with all permissions an app asks, even when the app functionality does not apparently need the requested permission [74].

Based on these assumptions, we consider the following app behaviour patterns as *suspicious*:

- An application loads dynamically the code that calls API functions protected with permissions. Indeed, malware can use this approach to run the code that could not be marked as suspicious by a static analyser because it does not call protected API.
- An application calls through reflection an Android API protected with a permission. This functionality can be used, for instance, to send malicious SMS, which cannot be detected by static analysis tools because the name of the SMS send function is encrypted and decrypted only at runtime.

Detection of these suspicious patterns has been added to our tool. StaDynA raises a warning if such patterns occur during the analysis. In a later section we show that analyzed malware families do indeed expose such suspicious behavior.

4.5 Android Class loading overview

The Dalvik Virtual Machine (Dalvik VM or DVM) allows a developer to exploit class loading functionality. This capability supports the developer need to load classes runtime from alternative locations such as internal storage or over the network [59]. This functionality in case of Android is usually used to:

1. Run applications that have more than 64K method references. Maximum number of method references in a dex file is 64K, but additional methods can be put in a separate dex file and loaded dynamically.
2. To extend the functionality of applications or frameworks at runtime with plug-ins loaded dynamically.

At the same time, although Android allows to load code dynamically and execute it, Google strongly recommends to avoid using this feature [34]. These recommendations are based on the fact that the Dalvik VM does not provide the secure environment for the code supplied dynamically. Thus, this code has the same permissions as the application

that loads this code. Moreover, the Dalvik VM does not isolate code from the underlying operating systems capabilities and, thus, dynamically loaded code can operate with native libraries without any constraints [34].

4.5.1 Android Class Loaders

Class loaders are responsible for controlling the loading of classes into the Dalvik VM. The process of loading classes in Android resembles the one implemented for the Java VM [105, 132]. As in the Java VM, there is also the *bootstrap* class loader responsible for loading core API classes. The *system* class loader is responsible for loading application classes. Additionally, an application may define additional class loaders to provide special ways of class loading.

In Android, as in Java, class loaders form a tree. To organize this structure, each class loader holds a reference to its parent. The *bootstrap* class loader is the root of this tree having a `null` reference to its parent. In Android all particular class loaders are derived from `java.lang.ClassLoader` (possibly indirectly). Android provides several concrete implementations of this class, `PathClassLoader` and `DexClassLoader` being the widely used ones.

4.5.2 Class Loading Process

When a class loader is asked to load a new class (`loadClass` method) the following steps are performed. At first, it performs a search to discover if this new class is already been loaded by the current class loader using `findLoadedClass` method. If the class is not found, the system tries to call `loadClass` method of the parent class loader (i.e., it tries to find the class loaded by the parent class loader). The process continues until the system reaches `BootClassLoader`, which calls the native method `loadClass` of the bootstrap class loader. If the class is not found by the bootstrap class loader `ClassNotFoundException` is generated. The exception is propagated back to the previous class loader in the tree. If it fails to find the class, `ClassNotFoundException` is rethrown. The process continues back till the class loader, which has been asked to load a class. If this loader fails to load the class, `ClassNotFoundException` runtime exception is released.

4.5.3 Android class loading peculiarities

In Java there are two class loader methods that allow to load classes: `loadClass` and `defineClass`. The former finds and loads a class and returns a Class object if it is found. The latter allows to define a class from a byte sequence. At the same time, in Android the `defineClass` method in the `ClassLoader` class is defined as `final`, which means it cannot be overridden in nested classes, and just throws an `UnsupportedOperationException` exception. Therefore, the `defineClass` method currently cannot be used in Android.

Additionally, there is a special class `DexFile` in Android, which methods are used to load code from a file. `DexClassLoader` and `PathClassLoader` indirectly use these methods to load classes into the process memory. At the same time, `DexFile` methods can be invoked directly without using a class loader. However, a reference to a class loader is still needed, but in this case class loader methods are not called directly.

4.6 Reflection

Vast majority of computers are based on Von Neumann architecture. According to this architecture program instructions and data are stored in the same memory, thus in the memory there is no difference between them. Thus, there are no obstacles to observe and manipulate program instructions as data. In the earliest computers there were no obstacles to treat program instructions as data. Thus, assembly language in these computers had reflection capabilities because it allowed programmers to make programs that could modify themselves. Later, in more high-level languages as C reflective ability disappeared and then became available only in languages that has reflection embedded in their type system. The first language that has this ability is 3-Lisp [131] that was described in 1982. After that reflection appeared in many different languages. In 1997 reflection became available for Java with the release of JDK 1.1 [25].

The ability of a program to manipulate as data something representing the state of the program during its own execution is called *reflection* [50]. According to [50] there are two variants of this manipulation: *introspection* and *intercession*. Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning.

4.6.1 Reflection usage in Android

Although Android is based on the DVM, the Android reflection API is almost the Java's one. The API is used to access class information at runtime, and use this information to create new objects, invoke class methods, change the values of data field members [140]. More precisely, in Android the reflection API is used for the following purposes [140]:

- *Hidden API method invocation.* The developers of the Android operating system may mark some methods as hidden. In this case, the declaration of these methods does not appear in the SDK library and, thus, is not available for application developers. At the same time, app developers, who want to use these undocumented features of Android, may use the reflection API to invoke them.
- *Access to the private API methods and fields.* During the compilation, the compiler ensures that the rules of access to fields and methods according to the specified

modifiers hold. Unfortunately, using the reflection API at runtime it is possible to manipulate with modifiers and, therefore, gain access to private variables and methods.

- *Conversion from JSON and XML representation to Java objects.* The reflection is heavily used in Android to automatically generate JSON and XML representation from Java objects and vice versa.
- *Backward compatibility.* It is advised to use reflection to make an app backward compatible with the previous versions of the Android SDK. In this case, the reflection is exploited either to call the API methods, which have been marked as hidden in the previous versions of the Android SDK, or to detect if the required SDK classes and methods are present.
- *Plugin and external library support.* In order to extend the functionality of an application, the reflection API may be used to call plug-ins or external library methods provided during runtime.

4.6.2 Reflection API

According to [50] there are two variants of this manipulation: *introspection* and *intercession*. This section considers introspection and intercession methods of the reflection API provided by the Android OS.

Introspection

Introspection considers the reflection API in Android which allows a program to observe and reason about its state and the context:

- *Retrieving Class objects.* To start inspecting a class, at first, it is required to obtain a Class object that describes the model of this class. There are several ways how to get this type of model. The most common is the call `Class.forName`, which allows to obtain a Class object giving the name of the class as a string. Additionally, it is possible to get a Class object given an object of this class (using `getClass` method); specifying class literal (adding `.class` extension to the class name or `.TYPE` for primitive type wrappers); using dependent classes (it is possible to infer Classes of the objects declared in a given class). Moreover, class loaders also have methods (`loadClass` and `defineClass`), which give a possibility to get a Class object. Additionally, Android provides its own way using `loadClass` method of `DexFile`.
- *Accessing Constructor objects.* Having a Class object it is possible to get access to the Constructor objects of this class using the `getConstructor` method or its counterparts `getConstructors`, `getDeclaredConstructor` or `getDeclaredConstructors`.

- *Accessing Methods.* Similarly to obtaining access to Constructor objects, it is possible to get access to Method objects. For instance, using the `getMethods` call it is possible to retrieve an array of all public methods of the Class object. Additionally, one of the most popular ways is to use `getMethod` call, which returns a Method object with the given name and parameter types.
- *Accessing Fields.* Equivalently, it is possible to get access to the Field objects using the calls `getField`, `getFields`, `getDeclaredField` and `getDeclaredFields`.

Intercession

Intercession considers the methods how an application using the reflection API is able to modify its own execution state:

- *Object creation.* Having a Class or Constructor object it is possible to create an object of the class by using `newInstance` call of the Class and Constructor objects respectively. It should be mentioned that using the former call it is possible to invoke only the default (zero-argument) constructor, while in the latter case it is possible to provide parameters into the constructor.
- *Method invocation.* Methods obtained through a Class object can be invoked using the `invoke` method. It is also possible to invoke static methods providing `null` as a first parameter. Otherwise, an object reference on which the method is called needs to be passed.
- *Modification of field values.* Using families of methods for getting (`get*`) and setting (`set*`) it is possible to get and set the values of class fields correspondingly.

4.7 Implementation

This section provides the implementation details of some key aspects of StaDynA. The detailed workflow of our system is shown in Figure 4.2. StaDynA consists of two main parts: a server and a client. The server part is responsible for static analysis, while the dynamic analysis is performed on the client part. The communications between the server and client parts are carried out using *Android Debug Bridge* (ADB), a standard tool for communications between an Android device and a computer. On top of standard ADB commands, we implemented the media for communication between the server and client parts.

The analysis of an application starts at the server side. A special program (considered in details in Section 4.7.1) looks for reflection and DCL occurrences in the code of the provided app. In case neither of them is found, our program builds a MCG of the app and exits. Otherwise, it starts the dynamic analysis on a device with the modified Android OS,

which constitutes the client part of StaDynA. The details of the client implementation are considered in Section 4.7.2.

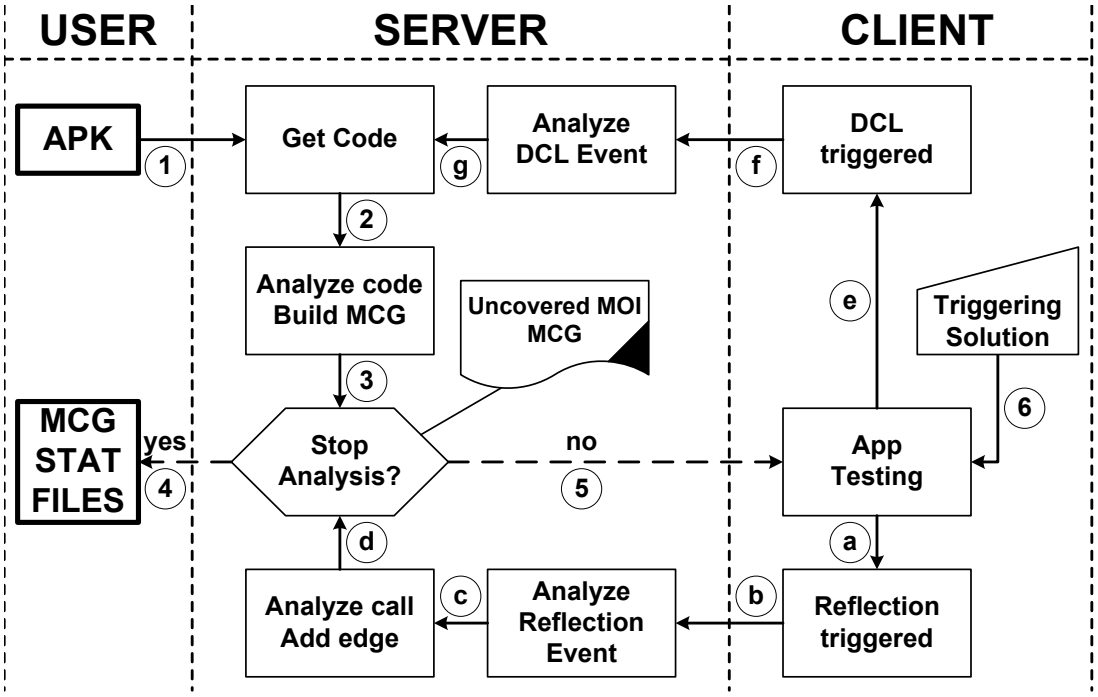


Figure 4.2: The StaDynA workflow

4.7.1 The server part

The server part of StaDynA is a Python program, which interacts with a static analysis tool. Currently, StaDynA uses AndroGuard [2] as a static analyser, which is a project encompassing a set of tools for static analysis of Android apps. AndroGuard represents compiled Android code as a set of Python objects that can be manipulated and analysed. However, StaDynA can work together with any static analysis tool that is able to analyse `apk` and `dex` files.

The server part of StaDynA starts the analysis of the provided app by extracting `classes.dex` file (see Step *1*, *2* and *3* in Figure 4.2), and then dissects the extracted code. The algorithm for the main steps is shown in Protocol 4. During this step StaDynA relies on the modified version of AndroGuard. The basic version of AndroGuard did not allow StaDynA to select all possible patterns of reflection and DCL calls. The updated list of patterns searched by StaDynA is presented in Table 4.4.

If MOIs are found StaDynA selects a device (real phone or emulator) to perform the dynamic analysis (Line 7) and then installs the app under analysis (Line 9) on the client side (see Step *5* in Figure 4.2). After that the server part obtains the UID of the installed package (Line 10) and starts a loop (Line 12 - 19) that analyses logcat messages

Protocol 4 App analysis main function algorithm

```
1: function PERFORM_ANALYSIS(inputApkPath, resultsDirPath)
2:     makeAnalysis(inputApkPath)
3:     if !containsMethodsToAnalyze() then
4:         performInfoSave(resultsDirPath)
5:         return
6:     end if
7:     dev ← getDeviceForAnalysis()
8:     package_name ← get_package_name(inputApkPath)
9:     dev.install_package(inputApkPath)
10:    uid ← dev.get_package_uid(package_name)
11:    messages ← dev.getLogcatMessages(uid)
12:    loop
13:        msg ← dequeue(messages)
14:        analyseStadynaMsg(msg)
15:        if finishAnalysis then
16:            performInfoSave(resultsDirPath)
17:            return
18:        end if
19:    end loop
20: end function
```

one by one. If a user finishes the analysis (Line 15) StaDynA saves the results (Line 16) and finishes its execution. If no MOI is found in the analysed application (Line 3 in Protocol 4), the server part of StaDynA saves the information about the analysed file (Line 4) and exits.

Table 4.4: The list of searched API calls

Class	Method	Prototype
Dynamic class loading		
Ldalvik/system/PathClassLoader;	*< init >*	.
Ldalvik/system/DexClassLoader;	*< init >*	.
Ldalvik/system/DexFile;	*< init >*	.
Ldalvik/system/DexFile;	*loadDex*	.
Class instance creation through reflection		
Ljava/lang/Class;	*newInstance*	.
Ljava/lang/reflect/Constructor;	*newInstance*	.
Method invocation through reflection		
Ljava/lang/reflect/Method;	*invoke*	.

Basically, each obtained message is represented in the JSON format and contains the values for the following fields: *UID* (required), *operation* (required), *stack* (required), *class* (optional), *method* (optional), *proto* (optional), *source* (optional), *output* (optional). The value of the `UID` field is used to obtain the messages produced by the installed app. An example of StaDynA message is shown in Listing 4.2

A function, which analyses the selected StaDynA messages obtained from the client, is implemented on the server. It extracts the value of the `operation` field and based on this value selects the appropriate routine to analyse the message. The function prototype is presented in Protocol 5.

The routines for the reflection messages analysis are similar, so we consider them on the example when operation corresponds to *reflection invoke*. The algorithm for analysis of the *reflection invoke* message is shown in Protocol 6. Lines 2 - 4 extracts the method name along with its class name and the prototype, which has been called through reflec-

```
{"uid":"10044", "operation":"2", "class":"Lcom/test/reflectiontestReflectionClass;", "method":"
    testReflection", "stack":["Ldalvik/system/VMStack;, getThreadStackTrace, (Ljava/lang/
    Thread;)[Ljava/lang/StackTraceElement;", "Ljava/lang/reflect/Method;, invoke, (Ljava/lang/
    Object;[Ljava/lang/Object;)Ljava/lang/Object;", "Lcom/test/
    reflectiontestReflectionMainActivity;, callReflectionFunction, ()V", "Lcom/test/
    reflectiontestReflectionMainActivity;, onClick, (Landroid/view/View;)V", "Landroid/view/
    View;, performClick, ()Z", "Landroid/view/View$PerformClick;, run, ()V", "Landroid/os/
    Handler;, handleCallback, (Landroid/os/Message;)V", "Landroid/os/Handler;, dispatchMessage
    , (Landroid/os/Message;)V", "Landroid/os/Looper;, loop, ()V", "Landroid/app/ActivityThread
    ;, main, ([Ljava/lang/String;)V", "Ljava/lang/reflect/Method;, invokeNative, (Ljava/lang/
    Object;[Ljava/lang/Object;Ljava/lang/Class;[Ljava/lang/Class;Ljava/lang/Class;IZ)Ljava/
    lang/Object;", "Ljava/lang/reflect/Method;, invoke, (Ljava/lang/Object;[Ljava/lang/Object;
    )Ljava/lang/Object;", "Lcom/android/internal/os/ZygoteInit$MethodAndArgsCaller;, run, ()V"
    , "Lcom/android/internal/os/ZygoteInit;, main, ([Ljava/lang/String;)V", "Ldalvik/system/
    NativeStart;, main, ([Ljava/lang/String;)V"], "proto":"(Ljava/lang/String;)V"}
```

Listing 4.2: An example of StaDynA message

Protocol 5 The algorithm of the client message analysis

```
1: function analyseStadynaMsg(message)
2:     msgOp ← message.get(JSON_OPERATION)
3:     if msgOp == MSG_REFL_INVOKE then
4:         processReflInvokeMsg(message)
5:     else if msgOp == MSG_REFL_NEWINSTANCE then
6:         processReflNewInstanceMsg(message)
7:     else if msgOp == MSG_DEX_LOAD then
8:         processDexLoadMsg(message)
9:     end if
10: end function
```

tion (invoke destination obtained from client `invDstFrCl`). Line 5 gets the stack from the message. Line 7 searches for the first reflection invoke call position in the stack. The next position corresponds to the method, which has called this reflection `invSrcFrStack` (Line 9). Then in the loop StaDynA compares this method with the list of MOIs extracted from the application executable (Lines 10 - 20). If the method is found StaDynA complements the MCG with the obtained information (Line 15), and deletes it from the list of uncovered invoke MOIs(Line 17). Otherwise, it adds this method to the list of suspicious methods (Line 21). This information is later analysed to see why the reflection calling method was not found in the application executable during static analysis phase. Similar procedure also occurs in the case if a call of a class constructor through reflection is performed.

The processing function for the DCL message slightly differs (see Protocol 7). From the message it obtains the source path of the file used to load the code (Line 2). Using this information, StaDynA downloads the file locally (Line 4), and processes it (Line 5). This process includes computation of the file hash and copying the file into the results folder with a new filename, which includes computed hash. The file hash allows us to check (in the function `fileAnalysed`, Line 14), which files have been already loaded, and thus, to avoid further extra analysis of already analyzed code. Otherwise, the code analysis for MOIs is performed for the loaded code (Line 15). Function `getDLPathFrStack` (Line 6) searches for a pair of a DCL call and a MOI in the stack corresponding to the "DCL" MOI extracted from the app executables. If this pair is found, then it is removed from

Protocol 6 The algorithm for analysis of the reflection invoke message

```
1: function processReflInvokeMsg(message)
2:     cls ← message.get(JSON_CLASS)
3:     method ← message.get(JSON_METHOD)
4:     prototype ← message.get(JSON_PROTO)
5:     stack ← message.get(JSON_STACK)
6:     invDstFrCl ← (class, method, prototype)
7:     invPosInStack ← findFirstInvokePos(stack)
8:     thrMtd ← stack[invPosInStack]
9:     invSrcFrStack ← stack[invPosInStack + 1]
10:    for all invPathFrSrcs ∈ sources_invoke do
11:        invSrcFrSrcs ← invPathFrSrcs[0]
12:        if invSrcFrSrcs ≠ invSrcFrStack then
13:            continue
14:        end if
15:        addInvPathToMCG(invSrcFrSrcs, thrMtd, invDstFrCl)
16:        if invPathFrSrcs ∈ uncovered_invoke then
17:            uncovered_invoke.remove(invPathFrSrcs)
18:        end if
19:        return
20:    end for
21:    addSuspiciousInvoke(thrMtd, invDstFrCl, stack)
22: end function
```

the list of uncovered DCL calls (Line 11). If the pair is not found, StaDynA adds the information about the dynamic code loading into the list of suspicious calls (Line 18).

Protocol 7 The algorithm for analysis of the DCL message

```
1: function processDexLoadMsg(message)
2:     source ← message.get(JSON_DEX_SOURCE)
3:     stack ← message.get(JSON_STACK)
4:     newFile ← dev.get_file(source)
5:     newFilePath ← processNewFile(newFile)
6:     dlPathFrStack = getDLPathFrStack(stack)
7:     if dlPathFrStack then
8:         srcFromStack ← dlPathFrStack[0]
9:         thrMtd ← dlPathFrStack[1]
10:        if dlPathFrStack ∈ uncovered_dexload then
11:            uncovered_dexload.remove(dlPathFrStack)
12:        end if
13:        addDLPathToMCG(srcFromStack, thrMtd, newFilePath)
14:        if !fileAnalysed(newFilePath) then
15:            makeAnalysis(newFilePath)
16:        end if
17:        return
18:    end ifaddSuspiciousDL(newFilePath, stack)
19: end function
```

Notice that the provided algorithms are simplified. For instance, in StaDynA the same MOI can in fact be used several times to call different functions or to load different code files (e.g., see the MCG in Figure 4.5, where the same "reflection invoke" node is used to call different methods).

4.7.2 The client part

The client part may be run either on a real device or on an emulator. Using the emulator is more convenient because one can run the client and server parts on the same machine. Moreover, it is possible to run several instances on the same machine and there is no need to attach several devices to a server. However, there can also be some drawbacks in using the Android emulator as a client side. The main drawback is that currently the Android

emulator is very slow. Moreover, mobile applications may suppress some functionality if they detect they are running in the emulated environment. With these limitations in mind, we implemented and tested our client part on a real device. However, the code is not device-dependent so it can be easily ported to an emulator or another device. As a reference device Google Nexus S smartphone is used. The modifications were added to the Android OS version 4.1.2_r2.

To obtain the information required for the analysis of reflection and DCL usage, we have modified the `DVM` and `libcore`. To obtain the information related to the DCL the method `openDexFile` of the `DexFile` class was modified. This method is called when a new file with the code is opened. It acquires three parameters as an input, two of which `sourceName` and `outputName` are of our interest. The added code forms a *JSON* message that contains the path to the file, from which the code is loaded (`sourceName`) and the path to the optimized version of the code (`outputName`). Along with this information, the stack trace data and the *UID* of the process are also added into the message, which then printed out to the *main* log file of Android.

To get the information about invocation through reflection, a hook was placed into the `invoke` method of the `Method` class. Each `Method` object has `declaringClass`, `name` and `parameterTypes` fields, which represent class name, method name and prototype of the invoked method respectively. This information along with the stack trace is put into the StaDynA message. Similarly, to log the information about new class creation through reflection, we put our hooks into the `newInstance` method of the `Class` and Constructor classes. However, in this case the name of the called method is fixed and equal to `<init>`. Moreover, the prototype in the first case is also fixed to value `()V` because the `Class.newInstance` method is intended to call the default constructor of a class.

Each StaDynA message contains the stack trace information. Stack trace is a sequence of method calls performed in the current thread starting from the most recent ones. The information from a stack trace is usually used to find the origin of an exception in a program. In our case, the stack trace information is used to detect the MOI, which calls the reflection or DCL methods. Basically, a stack trace is an array of stack trace elements. Each stack trace element contains information about the class name, the method name and the line number of the method in the sources. Unfortunately, using only this information without access to the sources of the app it is impossible to uniquely identify the MOI, because due to the function overloading it is possible when several methods in a class have the same name. Thus, to select the appropriate MOI from the stack trace, the information about the method prototype must be also added to a stack trace element, because the method name and its prototype allow to uniquely identify the method in the class. To obtain these data we modified `StackTraceElement` giving it a possibility to store the method prototype information (a separate field named `prototype`, which is

filled by our modified Dalvik VM).

A StaDynA message has a header and a body. To distinguish StaDynA from other log messages the client adds a special marker to the header. This marker allows the server to perform the preliminary filtering of messages leaving only the ones generated by the client. The second part of the message header is the part number. Currently, there is a limit in Android on the log entry size. The maximum size of a message is limited by the constant `LOGGER_ENTRY_MAX_PAYLOAD`. At the same time, a StaDynA message may exceed this limit. To overcome this problem, the client part of StaDynA split a message into several parts if it does not fit the maximum size of the log entry. The sequential number of a part is also added into the header, and is later used by the server to assemble the original message.

4.8 Method Call Graph

As a result StaDynA generates a MCG. Our implementation of MCG supports the visualisation of the peculiarities discovered during the work of StaDynA. Our tool facilitates the analysis of Android apps with dynamic features discovered during the analysis. We exemplify the capabilities of StaDynA on a simple application that exploits dynamic behaviour patterns, while the full description of graph components is given in Section 4.8.1. This application dynamically loads code and performs calls of the loaded methods through reflection. The MCG of the app before the analysis is shown in Figure 4.3, while the graph obtained with StaDynA is demonstrated in Figure 4.4.

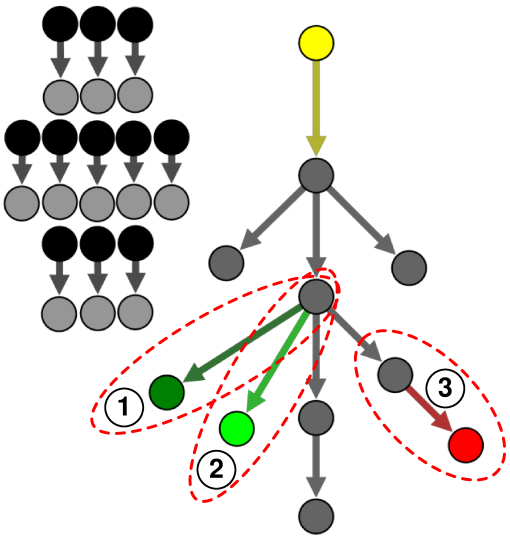


Figure 4.3: The MCG of the app before StaDynA

Figure 4.3 illustrates that after the static phase StaDynA selects 3 paths, which are

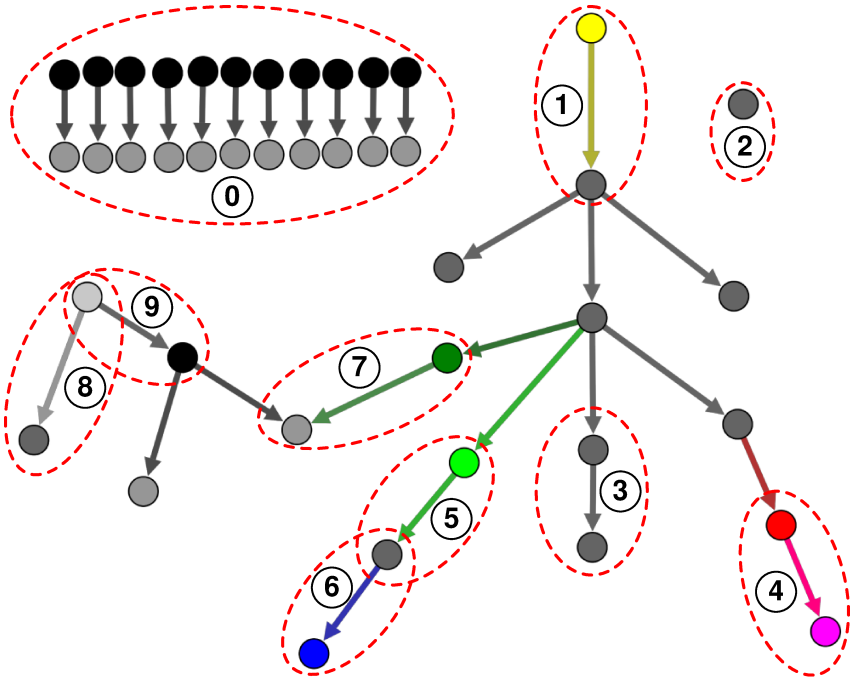


Figure 4.4: The MCG of the app after StaDynA

surrounded by dashed ellipses. Ellipse 1 shows that a MOI (the dark grey node) invokes a constructor (the dark green node) through reflection. Similarly, Ellipse 2 displays a method invocation through reflection. Ellipse 3 depicts that the DCL call (the red node) is performed in a method (the dark grey node). The description of all types of nodes and all possible edges is available later on in this section.

After StaDynA's analysis the graph is extended considerably (Figure 4.4). A number of additional nodes and edges appeared. During the dynamic analysis StaDynA added the edges that are outlined by Ellipses 4-7. Ellipse 4 shows that in the result of a DCL call (the red node) a new code file has been loaded (the pink node). The loaded code is analysed and the MCG is complemented with new nodes and edges from the loaded file. Ellipse 7 reflects that a class constructor (the grey node) is called through reflection. This ellipse depicts the case when a MOI has been resolved. Ellipse 5 shows a method invoked through reflection. However, the latter case differs from the previous one in the sense that the called method is attached with the blue node. The edge in Ellipse 6 indicates that the invoked method contains dangerous functionality protected by an Android permission. This kind of edges added at runtime allows us to raise a flag of suspicious behaviour. To obtain the map of API calls and corresponding permissions PScout [45] was used. In fact, AndroGuard provides similar functionality, but unfortunately, the map used in AndroGuard was obtained for Android 2.2 using [74], which can be considered obsolete now. We modified AndroGuard to use the map produced by PScout for Android 4.1.2.

Each node type is assigned with a set of attributes. The analysis of the values of these attributes may facilitate the dissection of Android applications having a complete method call graph obtained with the help of StaDynA. For instance, each method node is assigned with attributes, which correspond to class name, method name and signature of this method. A permission node is assigned with the permission level along with the information about the API call that it protects.

4.8.1 Method call graph description

All nodes used in our method call graph fall into two categories:

- ***Real nodes***. Real nodes represent real methods and functions, which constitute application.
- ***Fake nodes***. Fake nodes have been added to facilitate the analysis of Android applications, in particular with the help of StaDynA.

All nodes are grouped into several types. In particular, the following types constitute the category of *real* nodes:

- **Method nodes**. These nodes represent ordinary methods of a class and are colored in dark grey.
- **Constructor nodes**. These grey nodes correspond to constructor calls (`<init>`).
- **Static initialization blocks**. These nodes are light grey and they represent static initialization blocks, which are represented as a `<clinit>`) method of a class.
- **Reflection new instance nodes**. These dark green nodes are specifically added by StaDynA to represent the calls to object creation through reflection.
- **Reflection invoke nodes**. These light green nodes represent method invocation through reflection.

The following types of nodes are categorized as *fake*:

- **Fake dynamic code loading nodes**. These red nodes reflect DCL.
- **Fake dynamic code filename nodes**. Each node of this type corresponds to a code file loaded by an app at runtime and detected by StaDynA. These nodes are pink.
- **Fake entry point nodes**. An Android app may have several entry points: Activities, Services or Broadcast Receivers. Each entry point type corresponds to an equivalent node type in our MCG. The nodes corresponding to Activities are colored in yellow, those matching Broadcast Receivers are orange and Service nodes are khaki.

- **Fake class nodes**. The nodes of this type colored in black connect constructors and static initialization block of a class.
- **Fake permission node**. These blue nodes indicate the usage of permissions. A node of this color attached to a method means the method making an API call protected with a permission.

All edges between the nodes in our graph are directed. The meaning of an edge depends on the types of the nodes it connects. Here we consider different types of edges possible in our graph providing examples shown in Figure 4.3, 4.4 where possible.

- **Yellow (or Khaki, or Orange) -> Dark Grey**. This edge type shows a connection between Activity (or Service, or Broadcast Receiver) fake entry point node with the corresponding entry method of the app (e.g., see Ellipse 1 in Figure 4.4).
- **Dark Grey -> Dark Grey**. This the most widespread type of edges connects caller and callee methods. Ellipse 3 in Figure 4.4 depicts an example of this type.
- **Dark Grey -> Grey**. This connection shows that a method creates an object inside it (calls a constructor of the class).
- **Grey -> Dark Grey**. This edge means that a method is called from a constructor.
- **Black -> Grey**. The edge between these types of nodes connects a fake class node with a constructor node of this class. A number of examples of this edge type are shown in Ellipse 0 in Figure 4.4. The constructor nodes in this ellipse are attached to special classes in an Android app used to access the app resources, e.g., to drawables.
- **Light Grey -> Black**. This edge shows that a static initialization block node is attached to a class node (see Ellipse 9 in Figure 4.4).
- **Dark Grey (or Grey, or Light Grey) -> Light Green**. The edge shows that a method (or a constructor, or a static initialization block) performs method invocation call through reflection. The edge of this type is presented in Ellipse 2 in Figure 4.3. Basically, the node attached to the light green one represents the "invoke" MOI.
- **Dark Grey (or Grey) -> Dark Green**. The edge indicates that a method (or a constructor) performs an object instantiation through reflection (see Ellipse 1 in Figure 4.3). The node attached to the dark green one represents the "new instance" MOI.
- **Dark Grey (or Grey, or Light Grey) -> Red**. This edge depicts a method (or a constructor, or a static initialization block) calling a dynamic code loading method (a red node), e.g., see example in Ellipse 3, Figure 4.3. The node attached to the red shows the "dynamic class loading" MOI.

- **Dark Grey (or Grey, or Light Grey) -> Blue**. This edge demonstrates that a method (or a constructor, or a static initialization block) performs an Android API call, which is protected with a permission (blue node), for example, see Ellipse 6 in Figure 4.4.

As a result of StaDynA analysis the following types of edges may appear:

- **Light Green -> Dark Grey**. This link shows that a method (a dark grey node) is invoked through reflection (a light green node), e.g., see Ellipse 5 in Figure 4.4. If the called method is protected with a permission a link to a blue node will be also added (Ellipse 6).
- **Dark Green -> Grey**. This edge demonstrates that a constructor (a grey node) is called through reflection (see example in Ellipse 7). It should be mentioned that if the called constructor is protected with a permission, a link to a blue node will be also added.
- **Red -> Pink**. The edge shows a file with code (a pink node) being loaded using DCL (a red node). An example is shown in Ellipse 5 in Figure 4.4.

4.9 Evaluation

In order to evaluate StaDynA we tested it on real representative applications, both benign and malicious. The tests were run on a machine with 2.5 GHz Intel Core i5 processor and 4 GB DDR3 memory (the server side of StaDynA). Google Nexus S smartphone with the modified Android OS version 4.1.2_r2 was used as a client side connected to the StaDynA server using a standard USB cable. This section describes our test suite of benign and malicious applications and reports the results of our experiments.

The evaluation test suite consists of a set of 5 benign and 5 malicious applications. The benign applications were selected based on their popularity and presence of MOIs in the code. The malware samples were selected based on the study presented in Section 4.2 from the families exhibiting DCL as a part of malware behavior. Additionally, we selected two additional malware samples (`FakeNotify.B` and `SMSSend`) based on the reports of antivirus companies [73, 114]. Table 4.5 contains details of the 10 apps we used for experimentation with StaDynA.

To evaluate StaDynA, the selected apps were manually explored in order to provoke the behavior of interest. Table 4.6 presents the results of evaluation of the number of nodes and edges available before and after StaDynA analysis. Table 4.7 summarizes our findings and describes the effect of triggering different MOIs on the revised MCG. As StaDynA can uncover MOIs dynamically, we present both the total number of reflection

[1] In the Link {Hash} should be substituted with the value from the corresponding Hash field

Table 4.5: Description of apps used for evaluation

App	Description
Benign	
FlappyBird	A popular scrolling game
	Hash: a3e6958ce2100966f4e207778e4cdbe72788214148c7f4bfd042ba365498deb3
	Link: Pulled from Google Play Store now
Norton AV	A security app for Android
	Hash: 674f096f9c13da470c24ad15df246c3ac2a8ab0d01114f09c0dd08e14145265b
	Link: http://www.androiddrawer.com/8414/download-norton-security-antivirus-3-3-0-892-app-apk/
Avast AV	A security app for Android
	Hash: bc2c86a4c144e0c79f954b5ccd2921b44ce7176970e53ffd55f71fd21d54c7d7
	Link: https://play.google.com/store/apps/details?id=com.avast.android.mobilesecurity
Viber	An instant messaging (VoIP) app
	Hash: 60d34c2e8c5e5ffbfe5fef05fe781b3e901c2da86700b6b1446c88a706fc51f3
	Link: https://play.google.com/store/apps/details?id=com.viber.voip
Floating Image	A live wallpaper and floating image streaming app
	Hash: 07dcfac7df3b0d6b2d2927bea653b297d91460ba4496e9261e10d137492870f6
	Link: https://f-droid.org/repository/browse/?fdfilter=floating%20image&fdid=dk.nindroid.rss
Malicious[1]	
FakeNotify.B	Sends messages to premium numbers
	Hash: db92306dc09c3042f2344fd3169faa7f0c9fd4f03f88c026fbcead5b58bdfa84
	Link: https://www.virustotal.com/en/file/{Hash}/analysis/
AnserverBot	Uses DCL and takes periodic commands from a C&C server
	Hash: fd25d9e900e18eccdec93d843be374446274c8ace7a2a7de8e83697e933eddc8
	Link: https://www.virustotal.com/en/file/{Hash}/analysis/
BaseBridge	Sends premium rate messages, manipulates SMS and performs payments
	Hash: 8990b7592a5d0cd7482d3f6fac1226c826e601ffea6bc1a13b97424771559111
	Link: https://www.virustotal.com/en/file/{Hash}/analysis/
DroidKungFu4	Uses cryptographic functions, performs payments, sends SMS, accesses private info
	Hash: e35d338b2c4983b3597e73f42ff4bc3590e59e00b363a84f2995476ef28e4704
	Link: https://www.virustotal.com/en/file/{Hash}/analysis/
SMSSend	Manipulates and sends SMS, performs payments, accesses private info
	Hash: f15c84ffa9218b9785794ce2db01fb05a0e4968767d9aa6af0d530970e1477e4
	Link: https://www.virustotal.com/en/file/{Hash}/analysis/

calls and DCL calls found by StaDynAbefore and during the analysis ("Found"), and the initial number of MOIs in parenthesis ("Init."), i.e., available only in the initial apk file.

Table 4.8 presents the StaDynA's findings of new API calls protected by dangerous permissions (results are aggregated by permission name, actual number of new API calls is not presented). The column *New* is a checkbox showing that no API calls protected with this permission call were found in the original application, which, thus, can be considered as an overprivileged by the tools [45, 74].

4.9.1 Results on Benign Apps

`ImageView` does not contain dynamic class loading, and its MCG was not expanded significantly by StaDynA. A popular game `FlappyBird` contains 1 instance of DCL call that was successfully uncovered by our analysis, and several instances of Reflection Invoke and Reflection NewInstance (our analysis missed the latter), but the expansion of MCG produced by StaDynA was also relatively small (22 new nodes and 17 new edges[2]). More complex applications like the mobile antiviruses `Norton` and `Avast` and the popular

[2]Notice that this is a valid situation for dynamic class loading: not all added nodes were connected initially, and to connect them more reflective calls were required.

Table 4.6: Evaluation Results: Selected benign and malicious applications (Nodes and Edges)

	Nodes		Edges	
Apps	**Initial**	**Final**	**Initial**	**Final**
Benign Applications				
FlappyBird	8592	8614	11014	11031
Norton AV	42886	55372	65960	85665
Avast AV	31317	32363	43554	44956
Viber	42536	46312	60078	65627
ImageView	5708	5713	6488	6496
Malicious Applications				
FakeNotify.B	148	171	137	191
AnserverBot	1006	1614	1138	2093
BaseBridge	1172	1780	1364	2333
DroidKungFu4	1550	21168	1779	23589
SMSSend	431	537	826	951

Table 4.7: Evaluation Results: Selected benign and malicious applications (Reflection, DCL, Permissions)

	Refl. Invoke		Refl. NewInstance		DCL		Permission Nodes	
Apps	**Found (Init.)**	**Triggered**	**Found (Init.)**	**Triggered**	**Found**	**Triggered**	**Initial**	**Final**
Benign Applications								
FlappyBird	11 (10)	6	6 (6)	0	1 (1)	1	9	9
Norton AV	137 (18)	5	12 (8)	2	4 (4)	2	63	81
Avast AV	42 (42)	6	19 (19)	5	1 (1)	1	22	25
Viber	107 (101)	26	47 (21)	14	2 (2)	1	67	71
ImageView	6 (6)	5	2 (2)	2	0 (0)	0	7	7
Malicious Applications								
FakeNotify.B	68 (68)	68	9 (9)	9	0 (0)	0	1	2
AnserverBot	4 (4)	1	5 (4)	2	6 (5)	3	12	23
BaseBridge	5 (5)	1	3 (2)	2	3 (2)	3	14	25
DroidKungFu4	13 (9)	1	6 (4)	0	1 (1)	1	26	250
SMSSend	193 (193)	128	1 (1)	1	0 (0)	0	0	3

messenger `Viber` demonstrated significant expansion of their MCGs: more than 1000 of new nodes and edges were discovered by StaDynA for each app.

`Norton AV`, `Avast AV` and `Viber` also demonstrated suspicious behavior: they dynamically added code that invokes dangerous Android APIs protected by permissions. Notice that one of new API calls added by `Norton AV` (protected by the `WRITE_SYNC_SETTINGS` permission) was not even present in the original MCG. Thus, `Norton AV` would have been flagged as suspicious by a static analysis tool that identifies overprivileged apps (the ones that request more permissions than they actually use in the code). These examples show that benign applications may also exhibit suspicious behavior.

4.9.2 Results on Malware Samples

`FakeNotify.B` and `SMSSend` do not contain DCL calls, and new elements of their MCGs discovered by StaDynA appeared only as a result of reflection calls. Uncovered parts of MCGs of these apps are relatively small (while steal revealing hidden dangerous functionality). Even more interesting results are demonstrated by StaDynA on `AnserverBot1`, `Basebridge4` and `DroidKungFu43` where uncovered new parts of MCGs are comparable

Table 4.8: Evaluation: added dangerous permissions

App	Permissions	New
Benign Applications		
Norton AV	WRITE_SETTINGS	
	READ_PHONE_STATE	
	INTERNET	
	WRITE_SYNC_SETTINGS	v
	GET_TASKS	
Avast AV	INTERNET	
Viber	READ_PHONE_STATE BLUETOOTH	
	INTERNET	
Malware		
FakeNotify.B	SEND_SMS	v
AnserverBot	INTERNET	
	READ_PHONE_STATE	
BaseBridge	INTERNET	
	READ_PHONE_STATE	
DroidKungFu4	CHANGE_NETWORK_STATE	v
	ACCESS_COARSE_LOCATION	
	BLUETOOTH	v
	INTERNET	
	BLUETOOTH_ADMIN	v
	WRITE_SETTINGS	v
	SET_TIME_ZONE	v
	WRITE_SYNC_SETTINGS	v
	READ_PHONE_STATE	
	CHANGE_WIFI_STATE	v
	MODIFY_AUDIO_SETTINGS	v
	MOUNT_UNMOUNT_FILESYSTEMS	v
SMSSend	READ_PHONE_STATE	v
	SEND_SMS	v

in size with the original statically produced graphs. In fact, the `DroidKungFu43` code size exploded after dynamic class loading (an order of magnitude increase of the MCG size).

In fact `DroidKungFu43` loads the file `settings.apk` that contains approximately 13 times more nodes and edges than the original application. The other two malware samples where DCL is present are from the `AnserverBot` and `BaseBridge` families. Both samples contain more than one instance of DCL. These samples both load two files with the names `moduleconfig.jar` and `bootablemodule.jar`. The former one contains no MOIs, whereas the latter contains one instance of `Reflection Invoke` and `Dynamic Load` each. `bootablemodule.jar` then loads another file `mainmodule.jar` which does not contain any MOIs.

In contrast to the benign apps, all evaluated malware samples evaluation exhibit suspicious functionality. This is an interesting result, as it shows that advanced malware indeed tries to conceal its dangerous functionality and reveals these parts only at runtime. E.g., `SMSSend` did not have any node labelled with a dangerous permission prior to execution. StaDynA has uncovered 4 such nodes (new dangerous API calls are protected with permissions `READ_PHONE_STATE` and `SEND_SMS`).

Our results show evidence that malware samples are more over-privileged (they contain more permission types added dynamically), so it is valid to identify apps as suspicious if they are over-privileged. Yet, as benign apps can be overprivileged too, more research is

required to confirm or reject this approach, and StaDynA can be handy in exploration of this topic. Based on our testing on selected apps, we can conclude that StaDynA allows to uncover a significant number of new nodes and edges in MCGs even under manual app exploitation. The tool also assists the analyst by highlighting suspicious behavior (nodes corresponding to dangerous API).

Figure 4.5 exemplifies the results of StaDynA showing the MCGs of FakeNofify.B produced without and with StaDynA. Notice that the new green edges and one new blue node were identified and added by StaDynA.

4.10 Related Work

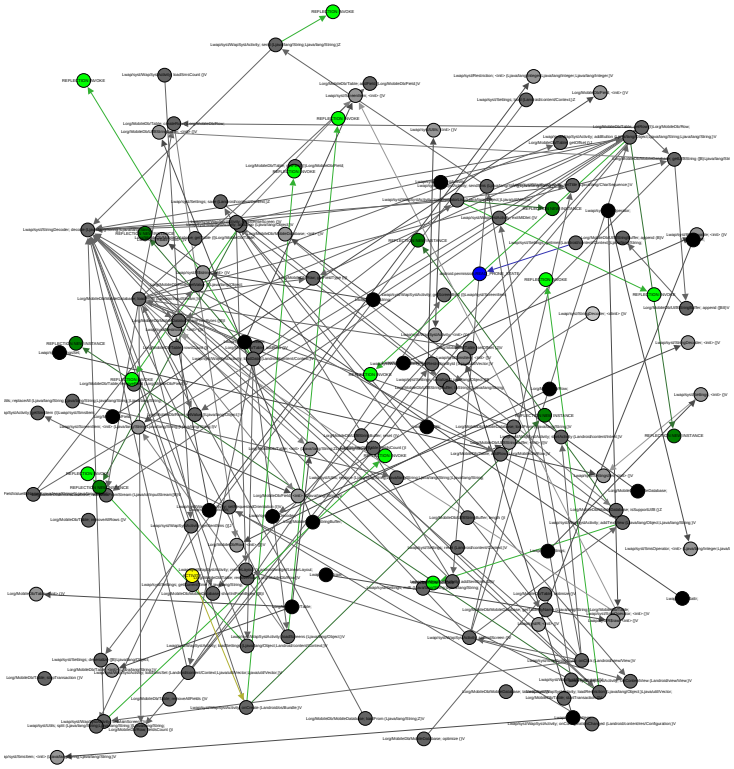
Being the most popular mobile OS, Android has won this position due to the openness of its ecosystem and the ease with which developers can publish apps on Google Play [22] and third-party markets. Yet the openness comes at the price of large volumes of malware apps polluting the ecosystem. One approach to tackle security and privacy of mobile apps is to extend the security controls of the platform to detect misbehaving apps or to enforce the desired security policy. The examples of such extensions are, e.g., FireDroid [126] that enforces defined security policies on interleaved systems calls; or the SELinux module for Android (SEAndroid [130]) that hardens the platform security and helps to prevent root exploits; but these approaches are out of the scope for StaDynA.

Another approach, which is more relevant to StaDynA, consists in analysis of the mobile app code. Many static and dynamic analysis techniques were already proposed for Android. The ded system [71] re-targets Dalvik bytecode into Java class files that can be analysed by the variety of tools for Java. In the original paper [71] the FortifySCA static analysis toolset was used for detecting vulnerabilities and dangerous functionality, like leaking the device IMEI.

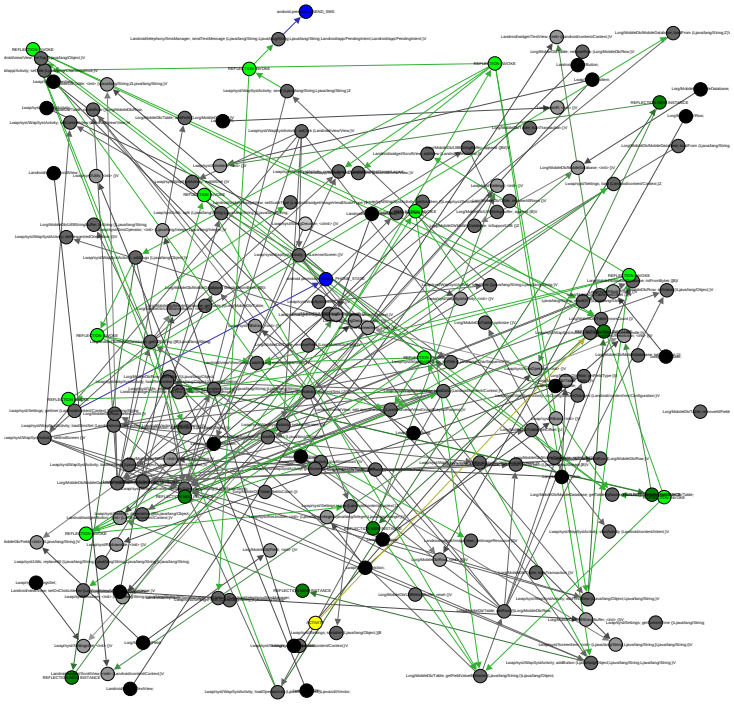
DroidAlarm [152] performs statical detection of privilege-escalation vulnerabilities in apps by constructing paths in inter-procedural call graphs from a sensitive permission to a public interface accessible to other apps. StaDynA complements these static analysis techniques by completing inter-procedural call graphs.

Hu et al. proposed to explore functional call graphs (FCG) and rely on graph similarity metrics to detect malware based on known malware graph patterns [95]. Gascon et al. continue this research direction for Android with a technique to detect malware apps based on comparing FCGs that are mined with AndroGuard [82]. StaDynA can complement these techniques by providing more precise graphs required for analysis.

TaintDroid was among the first dynamic analysis tools for Android apps [70]; it allows to manually track propagation of information via the TaintDroid infrastructure-equipped smartphone software stack. Sources of sensitive information are typically the device sensors or private user information, and sinks are network interfaces; thus the main scope of



(a)



(b)

Figure 4.5: FakeNotify.B MCGs: a) without StaDynA b) with StaDynA

TaintDroid is detection of privacy leaks. This approach is followed by DroidScope [145]. DroidScope allows to emulate app execution and trace the context at different levels of the Android software stack: at the native code level, at the Dalvik bytecode level, at the system API level, and at the combination of both native and Dalvik levels. While executing an app in DroidScope a security analyst can track events at different levels and instrument parameters of invoked methods to discover a malicious activity.

Dynamic analysis techniques are especially complicated to automate due to the need to emulate a comprehensive interactions of apps with the system and a user (UI interactions). Several approaches were proposed to automate the triggering of UI events, from random event generation [94] to more advanced approaches like AppsPlayground [120] and SmartDroid [151]. Our current version of StaDynA lacks for the automatic app exploration component like AppsPlayground or SmartDroid. We plan to integrate such kind of frameworks with StaDynA in the later versions.

Android apps are not the only carrier of malware distribution. Web applications also suffer from malware presence. Notice that the web code analysis techniques evolution took a similar path as on Android: first, static techniques were employed for detection of malicious code patterns; then dynamic techniques appeared as soon as the attackers had begun to obfuscate the malicious code; later the evasive malware that could change behavior in presence of a monitoring system has led to new hybrid techniques for evasive malware detection [101].

Recently, Poeplau et al. [117] have identified the problem of dynamic code loading in Android apps. The authors selected possible vulnerable patterns of dynamic code loading and built a tool that can analyze Android apps for the found patterns. Moreover, they also proposed to use a whitelisting approach to combat dynamic code loading that can potentially expose dangerous behavior. Whitelisting means that the loaded code should be vetted and included into the white list (or properly signed [147]) allowed to be run. However, as mentioned in the article [117], extraction of the dangerous behavior is a difficult problem by itself, especially when the protected API is called through reflection. In contrast, StaDynA aims not at preventing this loading (because a lot of legitimate apps use it and extra complications will not be welcomed by the developers) but at the analysis of applications in the presence of reflection and dynamic code loading. It is worth mentioning that StaDynA is more sensitive to DCL than the approach used in [117]: our modifications of AndroGuard allowed StaDynA to detect more instances (higher percentage) of class loading applications than in [117] (StaDynA reports that at least 18.5% of 13863 most popular apps in Google Play use dynamic class loading, while Poeplau et al. [117] found that 5.01% out of 1632 apps had this behavior).

Reflection and Dynamic Class Loading in Java. Gaps in the static analysis techniques in the presence of dynamic class loading, reflection and native code were previously studied

for Java. For example, similarly to our approach, in [92] a pointer analysis (based on program call graphs) technique for the full Java language is extended by addressing dynamic class loading and reflection via an "online" analysis, when a call graph is built dynamically based on the program execution, and dynamic class loading, reflection and native code are treated in real time by modifying the pointer analysis constraints accordingly.

A runtime shape analysis for Java was investigated in [52]. Traditionally a shape analysis operates based on the call graph of a program, and it allows to conclude how the heap objects are linked to each other (e.g., if a variable can be accessed from several threads). Yet in Java the call graph produced from a program can be incomplete; and [52] suggests how to execute an incremental shape analysis when the call graph evolves dynamically. Our proposal does not involve a shape analysis, yet the ideas behind our proposal and [52] are similar.

Livshits, Whaley and Lam have studied the reflection analysis for Java [106], and they have proposed to a static algorithm to infer more precise information on approximate targets of reflective calls, as well as to discover program points where user needs to provide a specification in order to resolve reflective targets.

One of the most similar to StaDynA works done for Java is authored by Bodden et al. [51] that proposes the TamiFlex tool that complements static analysis of Java programs in the presence of reflection and custom class loaders. Using the `java.lang.instrument` API TamiFlex modifies the original program to perform logging of class loading and reflection call events. This information is used to seed a tool, which performs static analysis of the program having the information obtained during the dynamic analysis phase. This work differs from StaDynA in several aspects. TamiFlex uses a special Java API that is not available in Android. Moreover, while it is possible to patch an app before loading it on a device, yet some malware already checks the signature in its code and if it does not correspond to the value in the code, it does not expose malicious behaviour. Thus, repackaging cannot be used in StaDynA. Also, TamiFlex requires some debug information (the line number of the function call), however, this information is not always present. That is why in StaDynA we modified the DVM itself to dump the prototype of a function, the functionality which is not available in a non-modified version.

We follow the same ideas on how to update dynamically program call graphs as in [51, 52, 91, 106], however we focus on different application domain (security analysis for Android apps) that also has a different security model than Java. Java sandboxes the loaded code while Android executes it in the same sandbox with the original application. Moreover, the loaded code on Android has a signature to identify its authenticity and integrity, so it is not possible to use the idea of [51] of modifying the app by inserting logging instructions. These peculiarities of Android security require new frameworks like StaDynA to provide better app analysis.

Chapter 5

Attestation Service for the Android Platform

In the Android ecosystem, the process of verifying the integrity of downloaded apps is left to the user. Different from other systems, e.g., Apple App Store, Google does not provide any certified vetting process for the Android apps. This choice has a lot of advantages but it is also the open door to possible attacks. To address this issue, this chapter presents how to enable the deployment of application certification service, we called TruStores, for the Android platform. In our approach, the TruStore client enabled on the end-user device ensures that only the applications, which have been certified by the TruStore server, are installed on the user smartphone. We envisage trusted markets (TruStore servers, which can be, e.g., corporate application markets) that guarantee security by enabling an application vetting process. The TruStore infrastructure maintains the open nature of the Android ecosystem and requires minor modifications to Android stack. Moreover, it is backward-compatible and transparent for developers, and does not change the application management process on a device.

The rest of this chapter is structured as follows. Section 5.1 describes why there is a need for an attestation service for Android. In Section 5.2 we overview TruStore, while in Section 5.3 we provide the implementation details of our solution. Section 5.4 discusses the implications of TruStore on application management process and on the Android platform and ecosystem. In Section 5.5 we discuss the existing approaches of securing mobile platforms and how TruStore fits them.

5.1 The Problem of Absence of Attestation Service Infrastructure for Android

During the last several years we observed an unprecedented growth of the Android ecosystem. Google tried to make the Android platform as open as possible: anybody can develop applications (or apps, for short) for it. Moreover, it does not tie developers to a specific

application market, providing them a possibility to freely publish their apps on third-party stores. This all leads to the sustainable growth of the number of third-party applications; moreover, there exist a number of third-party markets such as Amazon, Yandex and so on.

At the same time this popularity of the Android platform also attracts adversaries. E.g., Kaspersky Lab recently reported on 20.000 new Android malware samples detected in the beginning of 2013 [109]. Even if a user shops for apps only on the official Google Play market, she cannot be totally sure in malware absence on her device, as reported by security companies[1], as well as by security researchers [157]. Secondly, there is also a lot of examples of apps, which are not classified as malware, but at the same time they collect a lot of private information about the users [70], or are simply of poor quality.

Security companies have started to propose various solutions for securing mobile platforms: anti-virus applications, approval by certificate authorities [133], and reputation services integrated with markets [135]. Yet, these are not enough. Anti-virus software for smartphones in its current format is very limited (not only it drains the battery faster, but it will detect only the threats recognized by a security company, while ignoring, e.g., privacy leaks or application collisions); certificate authorities are known to be susceptible to failures [122]; and reputation services by nature cannot react quickly to new threats. Especially in the context of an enterprise that accepts the so-called Bring-Your-Own-Device (BYOD) paradigm these solutions are of limited applicability, as they do not allow a fine-grained control over security of apps installed on the platform.

Alternative solutions available could be summarized as follows: (a) app rewriting (e.g. [98,141]), (b) off-device app verification either by the user or on a market ([58,69,86]), (c) platform hardening (see for example [54,70,130]); or (d) a combination of those. However, they are not fully satisfactory for a variety of reasons. App rewriting is not acceptable from the legal perspective, as the rewriters (e.g. an enterprise information security staff or an end-user herself) are not the digital rights holders, and, therefore, rewriting is as legal as repackaging (a known source of app plagiarism and a malware distribution vector [85, 156]). Off-device verification and certification are quite efficient if done by the market. This approach somehow works for the Apple's walled garden ecosystem, even though Apple is known to ignore the user privacy concerns [42, 69, 116], but for Android apps this step is essentially missing. Google claims to check the apps submitted to Google Play, but the community has reported severe limitations of their app validation process, see, e.g., [112]. In the same time, as a consequence of the Android ecosystem openness, even if a third-party market will enable app verification process, there is no assurance that the installed app is the one that was checked on the market.

We aim to close this gap of lack of trust on a particular app market by enabling TruStore: a concept of a trusted market for Android that can guarantee security of

[1] http://countermeasures.trendmicro.eu/android-malware-believe-the-hype/

apps provisioned to end-users' devices. The idea of the TruStore implementation is based on the Android application signing process described in Section 2.6.3. TruStore adds an additional market signature to the vetted applications before provisioning. Then the end-user's device is responsible for checking whether the loaded application is signed by a trusted market signature.

In this chapter we propose an architecture of a trusted market and describe a proof-of-concept implementation for a regular Android device. We discuss how a trusted market may affect the most significant steps in the app lifecycle: installation, update, deletion and interactions with other apps and with the device infrastructure, and overview the potential of a trusted market for end-users, developers and enterprises with the BYOD policy enabled.

5.2 TruStore Overview

Let us overview our approach, which we have called TruStore. TruStore (stands for Trusted Store) provides an infrastructure for app distribution and management on an Android smartphone that can be trusted by a user.

The TruStore architecture consists of two main parts: a *server* and a *client*. The server part, besides offering the standard app provisioning functionality provided by app markets, is responsible for the application vetting process.

A possible vetting workflow can be the following. A developer uploads his application to the server. Along with the app package TruStore could also require the source code to ease the vetting process, like it is done for the Apple AppStore market. The server then checks the application for compliance with certain standards of secure applications. This process can include static analysis of application executables and source code (if provided) and dynamic analysis. During this process the server can also provide a short report about the functionality the application uses. This report can be later requested by users if they would like to understand better the application functionality and its usage of sensitive device features.

If the app has passed the vetting process, the server signs the application with its private key and places it in its market to be accessible by users. It should be mentioned that it is possible to sign an Android package without violating the integrity of the original developer's signature. This can be done in case the source code was not provided by the developer for some reasons. Notice that the server part implementation (including the app provisioning and vetting functionalities) is out of scope of this chapter work. The interested reader can refer to, e.g., [58, 69, 86, 157] for the examples of app verification frameworks.

The client part is based on a modified Android system; it allows the device holder to make use of TruStore: the user with TruStore protection enabled can install only the

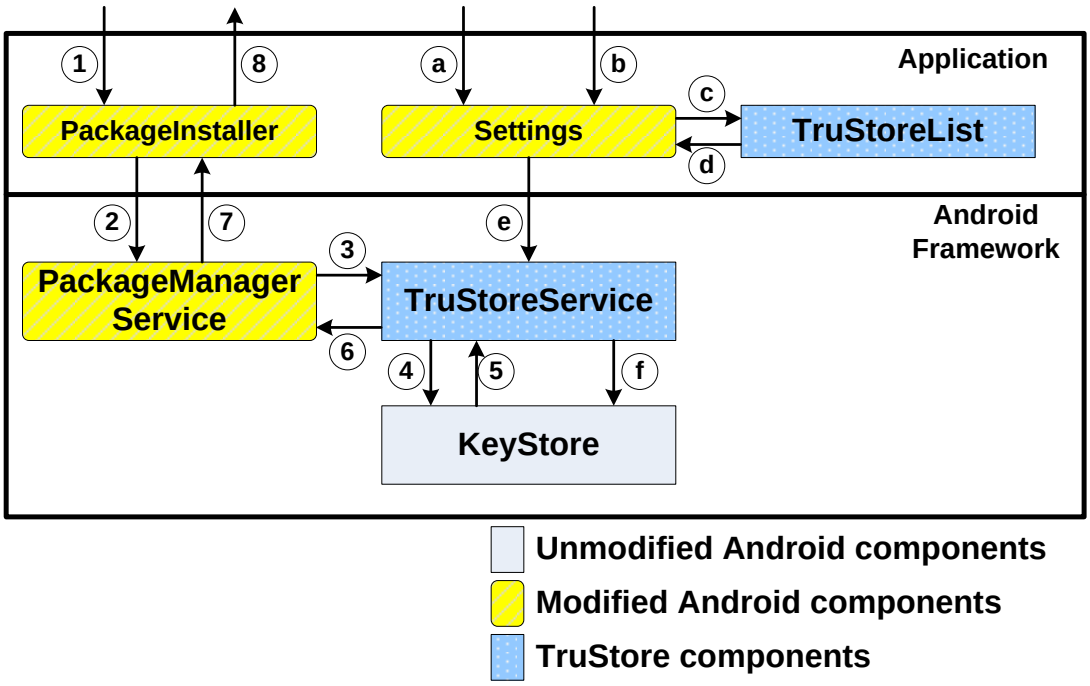


Figure 5.1: The TruStore architecture: the steps with letters represent the TruStore management process; those ones with numbers describe the checks during app installation.

packages signed with TruStore certificates. Figure 5.1 summarises the architecture of the client part of TruStore. The implementation intricacies of our system are presented in Section 5.3.

The TruStore management process starts with the activation of the TruStore protection. If a user wants to activate the TruStore protection she checks a special checkbox (Step *a* in Figure 5.1) added in the standard Android *Settings* application. After that she is able to select a special item (Step *b*) that will display the list of TruStore certificates installed in the system. On this screen the user may start the process of adding a new TruStore certificate to the system.

The `TruStoreList` application is responsible for displaying the list of certificates available to install on the external storage (Step *c*). The user can select a certificate and `TruStoreList` will pass the certificate back to the `Settings` application (Step *d*), which will store it in the system credential storage using `TruStoreService` (Steps *e* and *f*).

There are three main ways to install an application on Android:

- Using the *Google Play* application.
- Using the *PackageInstaller* application.
- Using the *adb* interface (`adb install` command).

These components interact with the `PackageManagerService` service responsible for

package management in Android. The service functionality related to the installation of new packages is protected with a special permission `INSTALL_PACKAGES`, which permission level is `signatureOrSystem`. This means that only the applications that are placed on the system image or signed with the Android platform certificate can communicate with this service to install a new package. Therefore, if the TruStore server uses a dedicated market application for distributing purposes it cannot directly start app installation using `PackageManagerService`. However, any app using a special intent can call the standard `PackageInstaller` application that can invoke the installation of a package and report about the installation process to the user.

The application installation process with the activated TruStore protection is presented in Figure 5.1. The user starts the installation using our modified version of `PackageInstaller` and performs the usual sequence of app installation steps (Step 1). The installer notifies `PackageManagerService` that it has to begin the installation of the package into the system. From this point `PackageManagerService` performs the job, while `PackageInstaller` waits for the installation report. In one of the checks of `PackageManagerService` we added a hook that communicates with `TruStoreService` and passes to it the list of certificates extracted from the installed application (Step 3). `TruStoreService` checks if at least one certificate in the obtained list matches the TruStore certificate installed in the system (Step 4). If a match is found then `PackageManagerService` finishes the installation and notifies `PackageInstaller` about the success, otherwise it generates a special error that is displayed to the user by the installer application (Steps 7, 8).

5.3 TruStore Implementation Details

In this section we describe the implementation details of the TruStore architecture considered so far. TruStore is developed on top of Android Open Source Project (AOSP) [38]. The proof of concept has been implemented for the Google Nexus S phone with 4.1.2_r2 version of AOSP. The implementation of TruStore touches two levels of the Android software stack: the *Application* and the *Android Framework* levels; at both of these levels our proof-of-concept implementation modifies the standard Android components, as well as adds new parts. During TruStore implementation we followed the objective to make as less intrusive modification of the platform as possible, using standard components where possible. The proposed modifications will not change the process of Android application development, but will require changes in the Android system code, which can be easily incorporated by Google in the future releases of Android.

The process of the TruStore management begins from the modified `Settings` application. In this application we added two preferences that activate the TruStore protection and allow the user to see the list of installed trusted store certificates. The

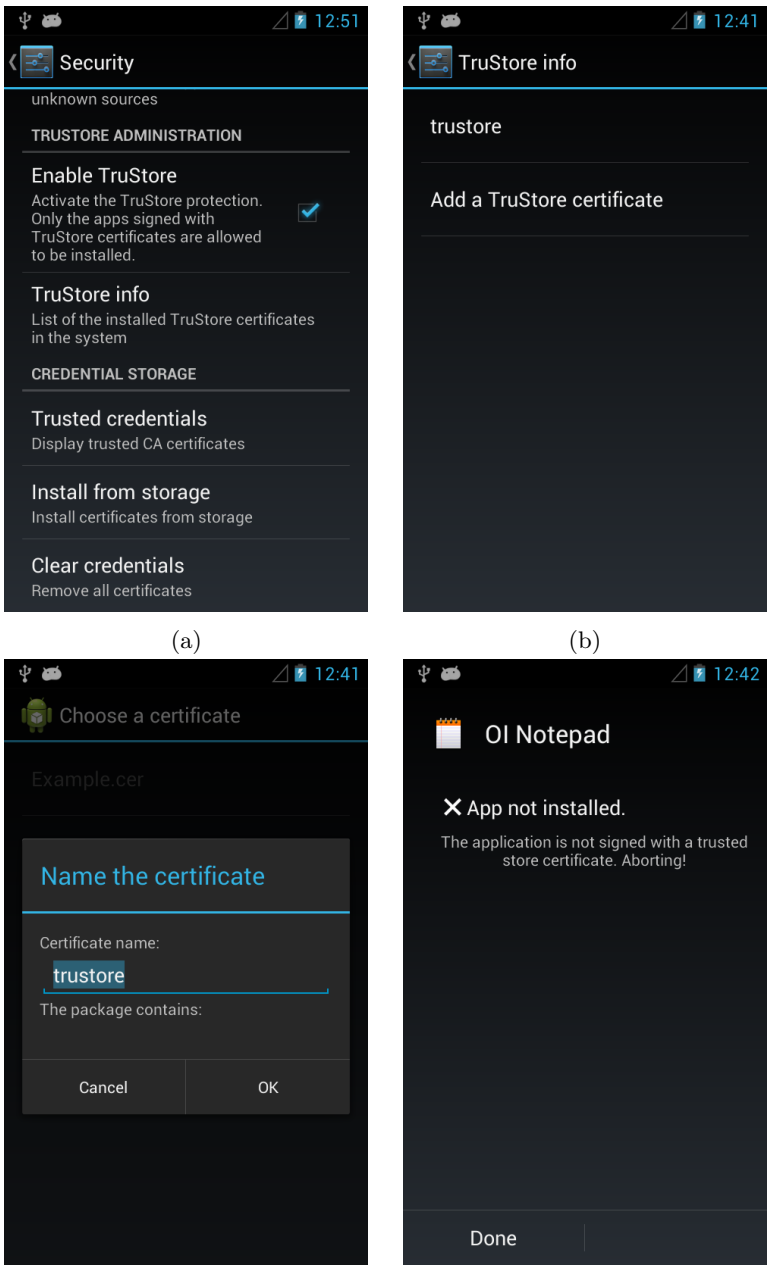


Figure 5.2: Screenshots of TruStore: (a) Settings to enable TruStore, (b) The certificate list of trusted stores, (c) TruStoreList application, (d) PackageInstaller error when a package is not signed by TruStore certificate

setting is written into `Settings.Secure` content provider and is later used by different components to detect if the TruStore protection is enabled. The screenshot of the modified `Settings` application with these added preferences is presented in Figure 5.2a, while in Figure 5.2b the list of currently installed trusted store certificates is shown. From this component it is possible to start the process of adding a new certificate.

As *Settings* application shares its UID with the system server and it is prohibited to read the content of the external storage from the system process, the functionality to search for and demonstrate the available TruStore certificates is extracted to an additionally implemented application *TruStoreList*, which is launched from the *Settings* app using an explicit intent. When the user selects new certificate she can provide additional information with it. Currently *TruStoreList* only provides the functionality to enter the name of the certificate, but other fields can be added to the dialog (shown in Figure 5.2c) with ease. For instance, along with a certificate it may require to save credentials, which can be used later to prevent unauthorised modifications of the stored trusted certificate. *TruStoreList* passes this information (the certificate and additional data) back to the *Settings* application, which in turn communicates with `TruStoreService` to store them.

`TruStoreService` (implemented at the *Android Framework* level) is responsible for storing and extraction of certificates and data related to them. It is also used to compare the list of obtained certificates with the stored ones (the positive result is returned if even one match is found). We decided to implement this functionality as a separate service in order to be able to add some additional features in the future. For instance, an external app, e.g., a corporate BYOD profile management app, may in the future be responsible for managing TruStore certificates having the interface provided by `TruStoreService`.

To preserve securely a certificate with additional data, the system *KeyStore* component is used. This is a standard way to store credentials on Android. This component automatically encrypts the stored information and grants access to it (based on UID) only to the component that originally initiated the preservation of the data. As `TruStoreService` is a part of the system server, only the Android components with the system UID equal to 1000 can read and modify these data. To distinguish the TruStore information from other credentials stored in the credential storage we add a special prefix (TRUSTORE). Thus, `TruStoreService` selects from the storage only the appropriate data. Additionally, `TruStoreService` may act as a cache for stored certificates.

To invoke a TruStore check we embedded a hook into the method `installPackageLI`. This hooks extracts the certificates of the installed package and compares them against the list of trusted store certificates using `TruStoreService`. If a match is found the installation process proceeds, otherwise `PackageManagerService` finishes the installation with a special TruStore error. The hook is embedded in the place when Android `PackageManagerService` has finished all verification steps. In this way, we are sure that the private keys corresponding to the certificates extracted from the package to be in-

stalled have been used to sign the package (thus, verifying the integrity of the signatures).

The last piece of the puzzle, `PackageInstaller` was modified to correctly display the explanation of the TruStore certification match error. An example of such fail is shown in Figure 5.2d. All other functionality of this component is left untouched. However, the correct handling of this error via, e.g., Google Play, cannot be guaranteed in our proof-of-concept implementation: as the Google Play application is not a part of AOSP, we cannot modify its sources. Yet, if the certificate check has failed, the user will still receive an error and the installation will fail.

To the best of our knowledge, the existing third-party Android application markets (e.g., Amazon) do not sign the deployed applications; therefore they do not take the responsibility for security of the deployed applications, while the TruStore does so.

5.4 Android Application Management with TruStore

As we mentioned before, the TruStore modifications have changed only the process of application installation. At the same time, due to the Android platform security architecture these changes also influence other aspects of application management. In this section, we consider how the process of application management has been changed with the TruStore modifications.

There are three main points in the lifecycle of each application: install, update and delete. The process of application deletion from the user or the developer point of view is not changed by TruStore, so it is not considered in this section. The installation process of an application is considered in details in the previous sections; it is not modified from the developer's perspective, but changes from the user perspective (installation may fail now).

Here it is worth mentioning that Android will prohibit the installation of the same application from different trusted stores, because these two applications have the same package name, but are signed with different sets of signatures. The same situation is for updates: an application cannot be updated by a store that has not installed it. A potential problem can arise if application updates in one market are vetted more quickly than in another. Thus, the user may want to install an update from another market, but will not be allowed to do this by the Android system.

At the same time, the TruStore approach does not reduce the app update capabilities of the developer, as the scheme of app distribution via a marketplace already assumes that the updates can be executed only through the market. In our approach the developer has to submit a new version of his app to the TruStore server, where it will be analysed, signed and distributed to users. The "kill switch" functionality available, e.g., on the Apple market and Google Play, which has proved very useful, can also be supported via TruStore.

The TruStore modifications influence the interactions with app components protected with `signature` and `signatureOrSystem` type of permissions, and the `sharedUserId` functionality for app interactions. The TruStore modifications will prohibit such kind of interactions between applications that are installed from different trusted markets, although they have been implemented by the same developer. This limitation comes from the fact that the applications installed from different markets will have different set of signatures (although developer signature may be the same). However, this restriction enables more security: TruStore will ensure safety of the apps loaded via itself, while safety of apps on other third-party markets cannot be guaranteed, and the interactions between trusted and untrusted apps may lead to information leaks and privilege escalation attacks.

5.5 Related Work

As we have mentioned, the existing approaches on securing a mobile platform generally fall into 4 categories: app rewriting, off-device app verification, platform hardening or a combination of these techniques. We now overview these approaches and discuss how a TruStore implementation fits with each of those. For the lack of space we do not discuss in detail the full body of work available for mobile platform security. Rather, we give examples of each technique, while analysing how TruStore relates with each particular approach.

Application Rewriting Application rewriting is a technique of changing the original code of an app e.g. by replacing calls to a sensitive APIs (e.g., the photo stream access) by the calls to a security controller introduced on the platform [46, 98, 141], which would dispatch these calls based on the desired policy. In this way retrofitted apps installed on the platform will not be able to stealthily grab the sensitive data.

By nature app rewriting is not suitable for iOS and other tightly controlled ecosystems, as it invalidates the original market signature. On Android the rewriting mechanism has to replace the developer's signature with a new certificate and enable a control over which app is signed with which certificate, in order not to break the app interactions established through custom developer signatures [141]. In our approach the trust relationships among apps are maintained automatically, as we do not remove the original signature of the developer. As well, as opposed to app rewriting, TruStore does not repackage the original application, therefore it does not violate the rights of the developer.

Application Vetting Application vetting (or off-device app validation) can be used either on a market (as it is done, e.g., for iOS) or by the user/an information security staff of a company before loading the application on device. The existing approaches

(e.g., [58, 69, 71, 86, 100]) aim at analysing the app code and retrieving security-relevant aspects of app behavior. For instance, Stowaway [58] analyses Android apps for vulnerable interaction patterns and PiOS [69] aims at detecting privacy violations, when an app illegally distributes sensitive user data. The app analysis can be static [58, 69] or dynamic [86, 100, 112].

The TruStore approach is by nature complimentary to application vetting. The server part of a trusted store is in fact expected to validate the app code submitted by developers.

Platform Hardening Improving the device security by introducing new components to the platform is currently the leading approach (if judged by the number of research papers). Notice that TruStore also falls in this category: we introduce new components of the Android middleware in order to implement the trusted certificates storage and execute the certificate checks.

There exist a lot of proposals of improvement of mobile platforms that tackle different security goals: for example, improvement of the permission system [78], enhancing the user control over her private data [62, 70] or protecting applications from malicious interactions and banning the privilege escalation [54]. The TruStore concept is limited in comparison with some of those techniques in the device context aspect. Namely, the frameworks for platform hardening at runtime are able to monitor the current situation on the device; they prevent the attacks because they have knowledge of each individual app and its actions. TruStore is not able to monitor the device security status, e.g., it is not able to prevent application collusions. Yet, TruStore achieves security by preventing loading of malicious apps.

Other Techniques

The crowdsourcing technique for enhancing trust on mobile platform is an approach evangelized by all app markets and investigated by security researchers (see e.g. [42,135]). The user is encouraged to read app reviews and install only the apps that have received the community approval. However, the deficiency of this approach is lack of assurance for new or updated apps. For example, adversaries could publish an innocent game which would receive a high feedback, and then push an update with a malicious payload.

BYOD

Techniques for the BYOD policies enforcement overlap with the approaches examined above, but their main goal is to allow an enterprise to secure its assets (corporate apps, frameworks, data or network), while allowing an employee to use her own device for interacting with these assets. TruStore can be used as a stepping stone for implementation

of a BYOD solution in a company, when each application loaded onto a devices for company environment is validated and certified by TruStore. For instance, this scenario can be implemented for the MOSES system that we describe in Chapter 6.

Chapter 6

Supporting Security Profiles in Android

Smartphones are very effective tools for increasing the productivity of business users. With their increasing computational power and storage capacity, smartphones allow end users to perform several tasks and be always updated while on the move. Companies are willing to support employee-owned smartphones because of the increase in productivity of their employees. However, security concerns about data sharing, leakage and loss have hindered the adoption of smartphones for corporate use. In this chapter we present MOSES, a policy-based framework for enforcing software isolation of applications and data on the Android platform. In MOSES, it is possible to define distinct *Security Profiles* within a single smartphone. Each security profile is associated with a set of policies that control the access to applications and data. Profiles are not predefined or hardcoded, they can be specified and applied at any time. One of the main characteristics of MOSES is the dynamic switching from one security profile to another. We run a thorough set of experiments using our full implementation of MOSES. The results of the experiments confirm the feasibility of our proposal.

The rest of this chapter is organised as follows. In Section 6.1 we recall the preliminaries that have driven the research presented in this part of the dissertation. In Section 6.2 we discuss the related work approaches proposed to address the problem. MOSES is presented in Section 6.3, while details of its architecture are discussed in Section 6.4. Section 6.5 covers MOSES implementation in details. In Section 6.6 we report on a thorough evaluation of MOSES.

6.1 Virtual Environments for Smartphones

Worldwide smartphone sales totalled 250 million units in the third quarter of 2013, up 46 percent from the same quarter of 2012 [20]. In the smartphone domain, the Android OS is by far the most popular platform with 82% market share. Those figures clearly show

the pervasiveness of Android, mostly justified by its openness to third party developers.

Smartphones allow end users to perform several tasks while being on the move. As a consequence, end users require their personal smartphones to be connected to their work IT infrastructure. More and more companies nowadays provide mobile versions of their desktop applications. Studies have shown that allowing access to enterprise services with smartphones increases employee productivity [12]. An increasing number of companies are even embracing the BYOD: Bring Your Own Device policy [40], leveraging the employee's smartphone to provide mobile access to company's applications. Several device manufacturers are even following this trend by producing smartphones able to handle two SIMs (Subscriber Identification Modules) at the same time.

Despite this positive scenario, since users can install third-party applications on their smartphones, several security concerns may arise. For instance, malicious applications may access emails, SMS and MMS stored in the smartphone containing company confidential data. Even more worrying is the number of *legitimate* applications harvesting and leaking data that are not strictly necessary for the functions the applications advertise to users [70, 84]. This poses serious security concerns to sensitive corporate data, especially when the standard security mechanisms offered by the platform are not sufficient to protect the users from such attacks.

One possible solution to this problem is *isolation*, by keeping applications and data related to work separated from recreational applications and private/personal data. Within the same device, separate *Security Environments* might exist: one security environment could be only restricted to sensitive/corporate data and trusted applications; a second security environment could be used for entertainment where third-party games and popular applications could be installed. As long as applications from the second environment are not able to access data of the first environment the risk of leakage of sensitive information can be greatly reduced.

Such a solution could be implemented by means of virtualization technologies where different instances of an OS can run separately on the same device. Although virtualization is quite effective when deployed in full-fledged devices (PC and servers), it is still too resource demanding for embedded systems such as smartphones. Another approach that is less resource demanding is paravirtualization. Unlikely full virtualization where the guest OS is not aware of running in a virtualised environment, in paravirtualization it is necessary to modify the guest OS to boost performance. Paravirtualization for smartphones is currently under development and several solutions exist (e.g., Trango, VirtualLogix, L4 microkernel [142], L4Android [67, 104]). However, all the virtualization solutions suffer from having a coarse grained approach (i.e., the virtualised environments are completely separated, even when this might be a limitation for interaction). Other limitation is the hardcoding of the environment specification. Environments cannot be defined by the user/company according to their needs but they are predefined and hard-

coded in the virtual machine. Furthermore, the switching among environments always require user interactions and it could take a significant amount of time and power. While researchers are improving some of these aspects [44], the complete separation of virtual machines and the impossibility to change or adapt their specifications remain an open issue.

This chapter presents MOSES, a solution for separating modes of use in smartphones. MOSES implements soft virtualization through controlled software isolation. Basically, MOSES provides a possibility to separate different aspects of user life activity specifying several *Security Profiles* on a smartphone. Such solution provides a user to use the same device in different environemts, e.g., work and personal, providing the control of the profiles to the interested parties.

6.2 Related Work

This section provides an overview of the related work. In particular, Section 6.2.1 describes research efforts in enhancing the security of the Android platform. Section 6.2.2 discusses BYOD approaches for mobile systems. Solutions based on secure container, virtualization techniques and other approaches are described in that section.

6.2.1 Android security extensions

There are a lot of solutions proposed to improve the security of Android. In this section we consider the ones that are more related to MOSES.

In Android, at installation time users grant applications the permissions requested in the manifest file. Android supports an all-or-nothing approach, meaning that the user has to either grant all the permissions specified in the manifest or abort the installation of the application. Moreover, a permission cannot be revoked at runtime. To circumvent this coarse-grained approach, several solutions have been proposed. Apex [111] allows users to select which permissions to grant to an application during the installation. Saint [113] is a policy-based application management system aiming at controlling how applications interact with each other. CRêPE [62] allows a user to create policies that can automatically control the granting of permissions during runtime.

More recently, [49, 158] concentrate on the protection of the user's private data. In particular, MockDroid [49] is a system that can limit the access of the installed applications to phone data by filtering out information. For instance, an application querying the contacts' provider may receive no results even if the provider is not empty. This approach is further refined in TISSA [158] where users are able to define the accuracy level of the information revealed to the application by means of privacy levels.

Taintdroid [70] proposes dynamic taint analysis to control how data flow between applications. In Taintdroid, taints are statically associated with predefined data sources,

such as the contact book, SMS messages, the device identifier (IMEI), etc. Taintdroid tracks the flow of tainted data and notifies the user if the tainted data leave the device through the outbound network connections. By using Taintdroid's tainting capability, AppFence [93] provides additional mechanisms to shadow sensitive data and to block unauthorised leakage of data via network. YAASE [125] encompasses tainting to prevent confuse deputy and privilege escalation attacks. In [76,77] Taintdroid capabilities are used to enforce data-driven usage control. In [43,102] taint tracking enables the system to trace sensitive information, enterprise and health data respectively, and enforce policies for that data. Unfortunately, Taintdroid has some limitations such as inability to trace implicit flows. Moreover, it prevents the load of shared libraries by third-party applications to prevent leakages through native code.

Context information plays a pivotal role to enhance security in mobile devices. In [47, 56,62,111], context is used to trigger security rules at runtime. The approaches in [76,77] use context to limit access to data in some environments. In [43], special context is a necessary condition to generate security notifications. In [102] the context is used to taint data generated in predefined environments. FlaskDroid [56] uses context to set up the values of one or more boolean variables in policies. These boolean variables are later used to instantiate a policy that is enforced by FlaskDroid's policy enforcement system, which is based on the extension of SEAndroid mandatory access control. The mandatory access control implemented in [56,130] considerably diminishes the effect of root exploits.

6.2.2 Bring Your Own Device approaches

Besides approaches to improve Android security in general, some solutions specifically aimed at supporting the BYOD have been proposed. The most important are listed below.

BYOD is an emerging paradygm, which allows employees to bring their own devices to work and to use them for execution of business related tasks. It is a win-win situation both for employees and companies. From business point of view this increases productivity and availability of the workers, from the other side it reinforces the satisfaction and happiness of employees [53]. At the same time, the use of personal devices increases security risks if they are connected to corporate network or have access to business data. To minimize the risks, BYOD policies have to exist in every organization backed with the abilities to control corporate data on the devices. One approach to control this is called *dual persona* [17] that provides possibility to maintain two separate environments on a phone (usually for personal and for corporate use), when the corporate one is controlled by IT department of a company. In this section we provide an overview of dual persona and other approaches to control the usage of personal devices for business tasks.

Secure container

Secure container (SC) is a special mobile client application that creates an isolated environment on the phone at the application layer. The application allows an enterprise administrator to create policies which control this isolated environment but cannot control the behaviour of a user outside this container [17]. This approach does not require the modification of the system image and is widely explored in the research community. AppGuard system [46] for instance, is a standalone Java application that disassembles apk file, inlines security checks before dangerous instructions according to a selected policy and then reassembles and signs the package. Thus at runtime, before executing a dangerous instruction AppGuard performs a security check and if the instruction is not allowed according to the policy an exception is thrown. Jeon et al. [98] use package rewriting to substitute dangerous instructions to equivalent ones, which are guarded by additional security checks. These guarded functions are implemented in a standalone Android service, which performs the additional checks. Aurasium system [141] intercepts some critical *Bionic* libc functions (e.g., `read()`, `write()`, `open()`) and calls the Aurasium (safe) version of them.

Many commercial solutions use the concept of security container implemented as a user application. NitroDesk TouchDown [30] and Good [21] offer solution with a prefixed set of business functionality in the container (i.e., email). Other solutions, (for instance, Fixmo [19]) offer a set of basic applications and also an SDK that can be used to develop new applications, if needed. The SDK provides wrappers for dangerous operations and passes them through the secure container application. Divide [16] and AT&T [13] use package rewriting technique to wrap up dangerous instructions of third-party applications, so that the interaction of these rewritten applications with the outer world happens through the secure container.

Mobile virtualization

Virtualization provides environments that are isolated from each other, and that are indistinguishable from the "bare" hardware, from the OS point of view. The hypervisor is responsible for guaranteeing such isolation and for coordinating the activities of the virtual machines. Virtualization has been widely used in traditional computers because it can: (i) increase security, and (ii) reduce the cost of deployment of applications (the hardware is shared in a secure way). With the spreading of mobile devices and with the increase of their performance capabilities the question of porting virtualization to mobile platforms became actual. Virtualization for mobile systems provides specific advantages like: (i) the possibility to separate communication subsystems (backed by real-time operating system) from high-level application code (which requires functional rich operating system with good interfaces); (ii) an opportunity to provide licence separation; (iii) a chance to

increase the security of the communication stack [90].

However, there are still several barriers for the adoption of virtualization in mobile devices. The main one is that ARM architecture, which is the most popular architecture for mobile devices, has a non-virtualisable instruction set architecture [104] (except Cortex-A15 design [110], which adds hardware-assisted virtualization capabilities). So as efficiency is a major concern in embedded virtualization, full virtualization approaches (emulation and binary translation) are not yet applicable for these devices because they are computational expensive. Thus, for embedded devices paravirtualization is used, which requires source-code modification of guest operating system [97]. There are several approaches to port popular Linux hypervisors to ARM architecture: Xen [97], L4 [104], KVM [65]. There are also several industry solutions: MVP by VMware [48], OKL4 by OK Labs [31] and vLogix Mobile by Red Bend [26]. All these solutions can be applied to create separate secure environments for business and private use. However, since all these virtual machines are simply ported to mobile platforms while being designed for PCs, they all share low performance.

A much better approach is Cells [44]. Cells is a new virtual machine specifically designed for mobile platforms. It provides lightweight virtualization for Android. The authors modified Android system in such a way that it is possible to have several separated environments, called Virtual Phones, based on the same operating system. Virtual Phones are completely separated from each other using kernel-level and user-level device namespace mechanism.

Yet, a common drawback to all the above solutions is that switching between virtual environments requires user interactions, and the configuration of each virtual environment is hardcoded and cannot be changed by the end-user.

Other approaches

Besides the above approaches there few other solutions. Gupta et al. [87] modified Android framework to support dual mode of operation, private and enterprise. The modification allows to restrict the use of communication capabilities of a phone, to force communication through enterprise VPN and have an encrypted external storage in enterprise mode. The authors of TrustDroid [55] proposed to monitor IPC communications, network traffic and filesystem access to separate data exchange between different domains, for instance, between enterprise and personal environments.

Other solutions use the capabilities of Taintdroid to track sensitive information. The main difference of these approaches is how to discover sensitive information. For instance, Feth et al. [76] proposed to rely on external authorities which supply data-usage policies with data. Thus, what data are sensitive in a smartphone is defined by external trusted authorities. In [102], the authors taint all the data that is produced or accessed by enterprise applications as sensitive information. Meanwhile, Ahmed et al. [43] relies

on the separation of public and private sources of data to detect sensitive information. Differently from MOSES, none of these solutions detects when a profile is active without user interaction. Furthermore, all of them offer only profiles predefined by the solution developers.

6.3 MOSES Overview

This section provides an overview of our approach named MOde-of-uses SEparation in Smartphones (MOSES).

MOSES provides an abstraction for separating data and apps dedicated to different contexts that are installed in a single device. For instance, corporate data and apps can be separated from personal data and apps within a single device. Our approach provides *compartments* where data and apps are stored. MOSES enforcement mechanism guarantees data and apps within a compartment are isolated from others compartments' data and apps. These compartments are called ***Security Profiles (SP)*** in MOSES. Generally speaking, a *SP* is a set of policies that regulates what applications can be executed and what data can be accessed.

One of the features introduced in MOSES is the automatic activation of *SP* depending on the context, in which the device is being used. *SPs* are associated with one or more definitions of ***Context***. A context definition is a boolean expression defined over any information that can be obtained from the smartphone's *Raw Sensors* (e.g., GPS sensor) and *Logical Sensors*. Logical sensors are functions which combine raw data from physical sensors to capture specific user behaviours (such as detecting whether the user is running). When a context definition evaluates to true, the *SP* associated with such a context is activated. It is a possible situation when several contexts, which are associated with different SPs, may be active at the same time. To resolve such conflicts, each *SP* is also assigned with a priority allowing MOSES to activate the *SP* with the highest priority. If *SPs* have the same priority, the *SP*, which has been activated first, will remain active.

MOSES permits a user to manually switch to a specific *SP*. To this end, MOSES provides a system app that the user can employ for forcing MOSES to activate a given *SP*. However, this behaviour can be restricted to avoid that the user activates unwanted *SP* in a given context (for instance, switching to a personal *SP* when at work).

Each SP is associated with an owner of the profile and can be protected with a password. A *SP* can be created/edited locally through an app installed on the device. Additionally, MOSES supports remote *SP* management. The former possibility may be used by a user of the phone for managing her personal *SP*, while the latter may be employed by an enterprise administrator to control the work *SP*. To avoid that the user tampers with the work *SP*, the security administrator protects the work *SP* with a password. In this way, MOSES can be used for realising a Mobile Device Management solution to manage

remotely the security settings of a fleet of mobile devices.

The current version of MOSES leverages the same idea of lightweigth separation of *SPs* as the one presented in [123]. At the same time, although the same idea is exploited, the approach used by MOSES is completely new. The previous version of MOSES [123] completely relies on Taintdroid to split data between different profiles. Data separation occurred using user-defined policies, which restricted the flow of information between different profiles. In the current version of MOSES, the separation of application data is implemented on Linux kernel level through filesystem virtualization approach. This allows our system to provide app data segregation out of the box. Moreover, a user in the new version of MOSES needs to define security policies only if she wants to apply fine-grained constraints to data.

6.4 Architecture

MOSES consists of the components presented in Figure 6.1. Central to MOSES is the notion of ***Context***. The `ContextDetectorSystem` component is responsible for detecting context activation/deactivation. When such an event happens, the component `ContextDetectorSystem` sends a notification about this to the `SecurityProfileManager`.

The `SecurityProfileManager` holds the information linking a *SP* with one or more *Context*. The `SecurityProfileManager` is responsible for the activation and deactivation of *SPs*. The `SecurityProfileManager` implements the following logic:

- If a newly activated *Context* corresponds to the active *SP* then the notification is ignored;
- If the *SP* corresponding to a newly active *Context* has a lower or equal priority to the currently running *SP*, then the notification is ignored;
- In all other cases, a *SP* switch has to be performed. This means that the currently running *SP* has to be deactivated and the new *SP* becomes active.

In the latter case, the `SecurityProfileManager` sends a command to the component `MosesHypervisor` informing what is the new *SPs* that needs to be activated.

The `MosesHypervisor` is the component that acts as a Policy Decision Point (PDP) in MOSES. The `MosesHypervisor` provides a central point for MOSES security checks against the policies defined for the active *SP* to regulate access to resources. The `MosesHypervisor` delegates the policy checks to its two managers: the `MosesAppManager` and the `MosesRulesManager`. The former is responsible for deciding which apps are allowed to be executed within a *SP*. The latter takes care of managing *Special Rules*.

The `MosesPolicyManager` acts as the Policy Administrator Point (PAP) in MOSES. It provides the API for creating, updating and deleting MOSES policies. It also allows

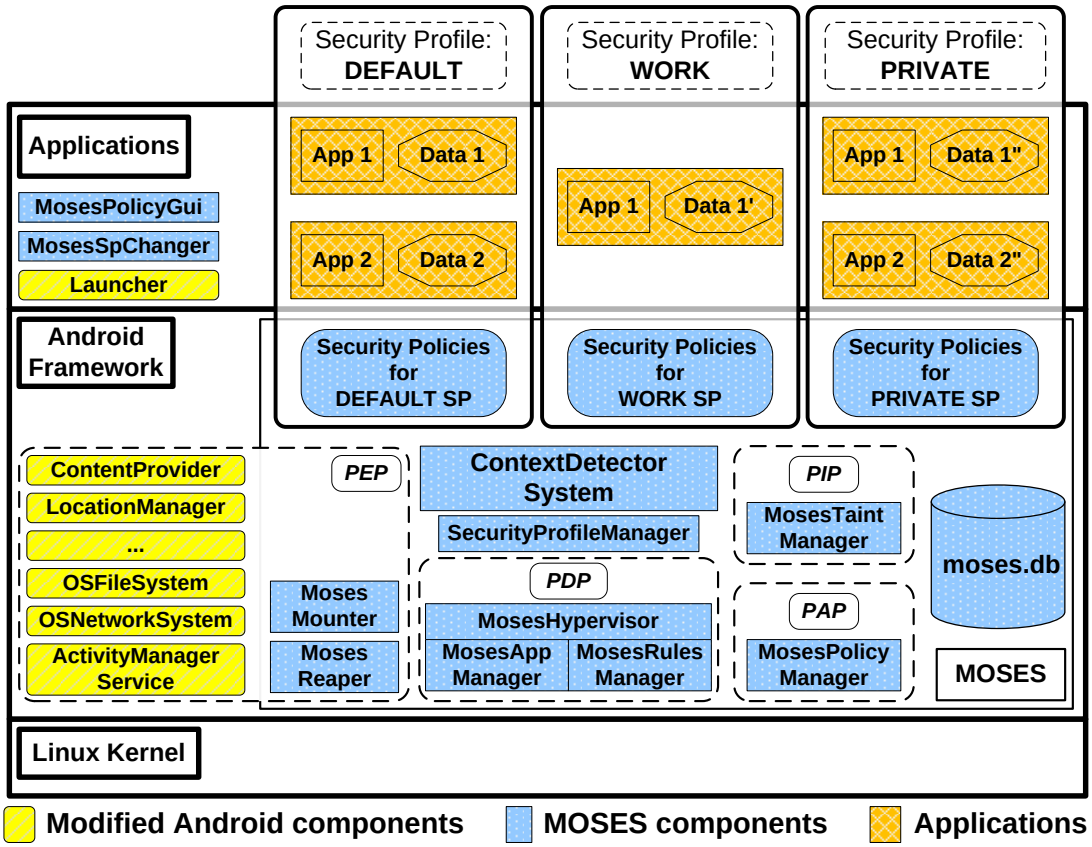


Figure 6.1: MOSES Architecture

a user to define, modify, remove monitored *Contexts* and assign them to *SPs*. Moreover, this component also controls access to MOSES policy database (`moses.db`) allowing only applications with special permissions to interact with this component. The permission has the type `signature`, therefore only the applications that are signed with the same certificate as the system will have access to the functionality of this component.

The `MosesTaintManager` component manages the "shadow database" which stores the taint values used by Taintdroid. We have extended the functionality of Taintdroid to perform more fine-grained tainting. In MOSES, we can taint specific rows of a content provider: to be able to perform per row filtering when an app access data in the content provider. For instance, it is possible to filter out from the query result data the rows which contain the information about device identifiers or user contacts. Given the fact that the enforcement of policies depends on the information provided by the `MosesTaintManager`, this component acts as a Policy Information Point (PIP).

The decisions taken by the `MosesHypervisor` need to be enforced by the Policy Enforcement Point (PEP). MOSES affects several components within Android middleware where decisions are enforced. For this reason, the PEP includes several Android compo-

nents such as `LocationManager` and `ActivityManagerService`. Moreover, some Android core classes (such as the `OSFileSystem` and `OSNetworkSystem`) are modified to enforce decisions regarding the access to the filesystem and network, respectively.

The enforcement of separated *SPs* requires special components to manage application processes and filesystem views. When a new *SP* is activated, it might deny the execution of some applications allowed in the previous profile. If these applications are running during the profile switch, then we need to stop their processes. The `MosesReaper` is the component responsible for shutting down processes of applications no longer allowed in the new *SP* after the switch.

In MOSES, applications have access to different data depending on the active profile. To separate data between profiles different filesystem view are supported. This functionality is provided by the `MosesMounter`. More details are considered in Section 6.5.2.

To allow the user of the device to interact with MOSES, we provide two MOSES applications: the *MosesSpChanger* and the *MosesPolicyGui*. The *MosesSpChanger* allows the user to manually activate a *SP*. It communicates with the `MosesHypervisor` and sends it a signal to switch to the profile required by the user. The *MosesPolicyGui* allows the user to manage *SPs*. We consider this component in details in Section 6.5.5.

6.5 Implementation

This section describes implementation details of some key aspects of MOSES. In particular, the version described here is based on the Android Open Source Project (AOSP) [**?**] version 2.3.4_r1. Moreover, MOSES incorporates the functionality of Taintdroid [70] to taint sensitive data.

6.5.1 Context Detection

One of the contributions of MOSES is that it can automatically switch *SPs* based on the current *Context*. The `ContextDetectorSystem` is responsible for monitoring *Context* definitions and for notifying the listeners about the activation or deactivation of a *Context*. The `SecurityProfileManager` component, which is one of these listeners, is notified about the change through the callback functions `onTrue(context_id)` and `onFalse(context_id)`, which correspond to activation and deactivation of a *Context* respectively. The `context_id` parameter represents a *Context* identifier. So as MOSES context detection functionality is decoupled from the rest of the system, it may be easily extended by integrating other context detection solutions [103, 136].

When the system starts up, MOSES selects from the database information about all *Contexts* and corresponding *SPs*. MOSES preserves this information in a runtime map in the form of $\langle C_i, (SP_k, prt_k)_i \rangle$, where C_i is the identifier of *Context* and $(SP_k, prt_k)_i$

is a tuple, which corresponds to the *Context* C_i and consists of *SP* identifier SP_k and the priority prt_k that corresponds to this profile. When the `ContextDetectorSystem` detects that a *Context* C_i becomes active (meaning the *Context* definition is evaluated to true), we select from this map the corresponding tuple $(SP_k, prt_k)_i$ and put it in the list of active *SPs*. Because more than one *Contexts* might be active at the same time, there may be more than one *SP* to switch to. In this case, from the list of active *SPs* the one with the highest priority is selected. If the selected *SP* identifier differs from the identifier of the currently running *SP*, the `ContextDetectorSystem` sends a signal to the `MosesHypervisor` to switch to the new profile. Similarly, when `ContextDetectorSystem` detects that a *Context* C_i becomes inactive, the tuple $(SP_k, prt_k)_i$ is deleted from the list of active *SPs*. After that the selection procedure of a *SP* with the highest priority is repeated.

6.5.2 Filesystem Virtualization

To separate data between different *SPs*, we use a technique called *directory polyinstantiation* [39]. A polyinstantiated directory is a directory that provides a different instances of itself according to some system parameters. In brief, for each *SP* MOSES creates a separate mount namespace [29].

The Android filesystem structure is quite stable, i.e., the system forces an application to store its files in the application's "home" directory that is `/data/data/<package_name>/` (`<package_name>` is the package name of the application). During the installation of an application, Android creates this "home" folder and assigns it Linux file permissions to allow only the owner of the directory (in this case the application) to access the data stored in it. To provide applications with different data depending on a currently running *SPs*, polyinstantiation of "data" folder may be used, i.e., for each *SP* a separate mount namespace, which points to different "physical" data folder depending on the identifier of a *SP*, may be created. In MOSES the described approach is used with two modifications. The first modification let the system to store all "physical" data directories under one parent directory (`/data/moses_private/`). The second modification creates the bindings not between the whole data folder and its "physical" counterpart, but bindings for separate application folders. The former modification allows MOSES to control direct access to the "physical" directories, while the latter permits to decrease storage overhead, because the usage of some apps is prohibited in some *SPs*.

The `MosesMounter` component is responsible for providing the above functionality. In particular, it receives the list of applications' package names that are allowed to execute in a *SP*. For each package name, the MOSES system builds the paths to the application "home" directory and to its MOSES "physical" counterpart, using the information of the identifier of a newly activated *SP*. These two paths are passed to the *mosesmounter*

native tool. This tool at first checks if MOSES "physical" directory exists. If not, then it creates this folder and copies there the initial application data from the corresponding "home" directory. Then the *mosesmounter* mounts the "physical" directory to a "home" directory using the Linux command `mount(target, mount_point, "none", MS_BIND, NULL)` [29], where `mount_point` corresponds to the path of the "home" directory and `target` corresponds to the path of the "physical" folder. Thus, the "home" directory always contains the initial copy of the application data, which are created by the Android system during the application installation. If a new *SP* is created, these initial data are copied to the "physical" directory providing the application with a fresh copy of its initial data as if the application has just been installed. If this process finishes successfully, the `MosesMounter` stores the name of this package in the list of mounted points. Thus, the process of polyinstantiation is completely transparent for the applications: after the mounting the applications work with the same paths as usual, although these paths point to another "physical" locations. Thus, there is no need to modify the applications to support the separation of data between different *SPs*.

Before switching to a new *SP*, the `MosesMounter` has to unmount all previously mounted points using the values stored in the list of mounted points. Similarly to the mounting, the `MosesMounter` passes the path to a mounted point (from the list of mounted points) to the *mosesmounter* tool, which performs unmounting. During this operation it is possible that some processes hold some files opened. In this case, the `unmount` command will fail. To overcome this problem, MOSES sends a `SIGTERM` signal to the process and repeats the unmounting. If after this the unmounting is still unsuccessful, the MOSES will send a `SIGKILL` signal to the process and once again performs the unmount operation.

6.5.3 Dynamic Application Activation

Each *SP* is assigned with a list of application *UIDs* that are allowed to be run when this profile is active. As it was discussed in Section 2.3, each application during the installation receives its own *UID*. MOSES uses these identifiers to control which applications can be activated for each *SP*. It should be mentioned that some packages can share the same *UID*. This happens if the developer of these applications have explicitly assigned the same value to `sharedUserId` property in the manifest files of the applications, and signed these packages with the same certificate. Thus, during the installation of these applications, the Android system assigns them the same *UID*. In this case, MOSES cannot distinguish these applications and if one of them is allowed in one profile the other will be allowed as well.

During the *SP* switching, the `MosesAppManager` selects from the MOSES database the list of *UIDs*, which are allowed in the activated profile, and stores it into the set of allowed `UIDs`. To control the launch of applications' services and activities, the

hooks into the `retrieveServiceLocked` and `startActivityMayWait` methods of the `ActivityManagerService` and the `ActivityStack` classes correspondingly are put. These hooks communicate with the `MosesAppManager` and check against the set of allowed apps if a component of an application can be launched. Additionally, the `MosesAppManager` controls the appearance of application icons in Android's Launcher application. When a new *SP* is activated, only the icons of the allowed applications for this profile will be displayed.

6.5.4 Attribute-based Policies

Within each *SP*, MOSES enforces an Attribute Based Access Control (ABAC) model [146]. The idea is that within each *SP*, users can define fine-grained access control policies to constraint application behaviour. For instance, the user may want to deny an application to read the files on an external storage. In this case, the user may write a policy which will still let the application to run within the profile but the access of this application to files on an external storage will be limited. For defining and editing policies, MOSES provides an activity shown in Figure 6.2d.

We have defined a simple policy language using the ABAC model. The attributes types that are taken into consideration in the MOSES language are capitalised in Listing 6.1. These are *Subject*, *Operation*, *Taint*, *Target*, and*SP-Name*. These attributes are described in the following.

```
1 Subject Operation [Taint] Target
2 decision [perform action(param-list)]  with scope SP-Name
```

Listing 6.1: Policy language used for ABAC rules

Subject represents the application to which the rule is applied. The application UID is retrieved through the GUI provided by MOSES for management (see Figure 6.2c), listing the applications installed in the system.

Operation is the action that the subject is executing. The value of this attribute is dependent on the the control hooks added by MOSES in the Android framework. Each hook communicates with a special class in the `framework` library that processes the information obtained from the operation hook. For instance, for controlling access to ContentProvider, we have injected hooks in the `ContentResolver` class. Similarly, for network and filesystem operations we have injected hooks in the `core` library. Currently, MOSES supports the following *Operation* types:

- **ContentProvider:** Query ContentProvider, Insert in ContentProvider, Update ContentProvider, Delete ContentProvider.
- **LocationProvider:** Get Last Known Location, Request Location Updates, Add

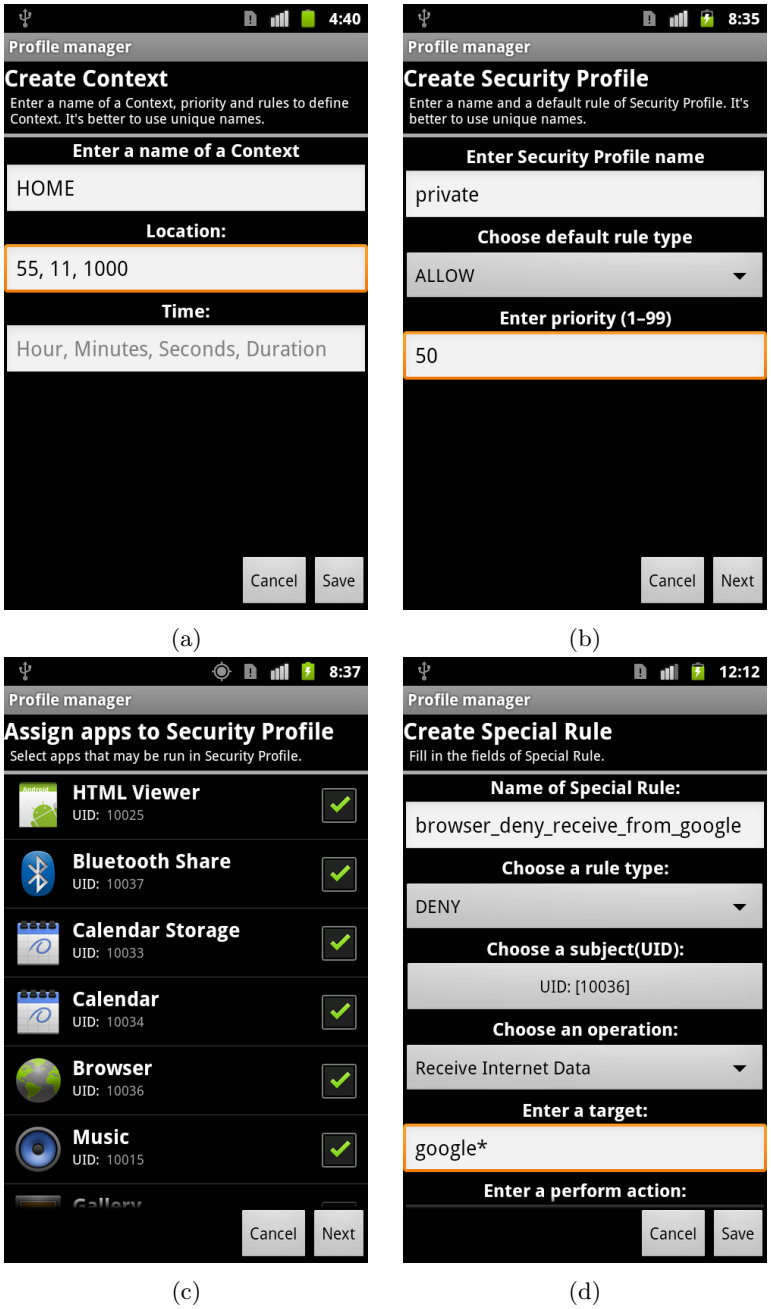


Figure 6.2: Screenshots of MOSES Profile Manager application: (a) Context creation, (b) Security Profile creation, (c) Application assignment to a Security Profile, (d) ABAC Rule creation

Proximity Alert, Request Single Update.

- **Network:** Receive Internet Data, Send Data to the Internet.
- **Filesystem:** Read from a File, Write to a File.
- **DeviceId:** Get Device ID.

MOSES supports also information flow control using the tainting mechanism provided by Taintdroid. Policies can include the optional attribute *Taint* to specify the taint type associated with the data accessed by the subject.

The *Target* attribute represents the resourced that is being accessed. It can have either fixed or volatile values. Values such as **LocationProvider** and **DeviceId** are fixed and correspond to `GPS` and `IMEI`. In the case of **ContentProvider**, **Network** and **Filesystem** the *Target* values are volatile. This means that a target may be specified partially. For instance, for the Filesystem the user can specify the following partial target `[/data/data/<package>/*]`. We developed these volatile targets to allow the system to enforce different behaviour for the targets that differs only partially. If a user wants to specify different access behaviour, for instance, for a directory and its subdirectory, she should create a separate policy rule for the directory and another one for the subdirectory.

SP-Name attribute represents the SP name where the policy is valid.

The decisions `ALLOW`, `DENY` and `ALLOW_WITH_PERFORM` can be assigned to policy rules. The effects of the first two are obvious. The decision `ALLOW_WITH_PERFORM` corresponds to allow with a restrictive obligation that performs additional action of the data returned by the operation. For instance, for "Get Device ID" operation a function can be chosen that will obfuscate the real `IMEI` of the device. The functions and their implementations are specified in a special built-in library. `ALLOW_WITH_PERFORM` decision manage to enforce security constraints specified in the profile minimizing the impact on already installed applications.

It is possible that two or more rules may be defined for the same attribute values. To resolve these conflicts, the user should also assign a priority value to each rule. In this case, the decision of the rule with the highest priority will have precedence over the decisions of other rules; in the case of equal priorities, then the last inserted rule takes priority.

For some combinations of attribute values, it might be the case that no rules apply. In this case, our system uses a default decision value (either `allow` or `deny`), which is assigned to the *SP*.

6.5.5 Security Profile Management

To give a user the ability to manage the *SPs* in her device, the *MosesPolicyGui* application is developed. This is a system application signed with a system key and assigned with

a special permission. This allows *MosesPolicyGui* application to communicate with the `MosesPolicyManager` and manage the *SPs*. Figure 6.2 provides several screenshots of the application running on a device. Due to the lack of space, we will not show screenshots of all activities the application provides.[1]

The *MosesPolicyGui* manages *Contexts* and *SPs*. We develop an application that allows a user to easily configure MOSES functionality. Figure 6.2a shows how to create a new *Context* definition. The user specifies the name of a *Context* and the parameters of the sensors used to detect the context around the device.

To define a new *SP* the application provides a wizard that guides the user through the steps. Figure 6.2b shows the first activity of this wizard. In this activity, the user has to define the name of a profile, the default decision and the profile priority. The screenshot in Figure 6.2c shows how to assign applications to the *SP*. Finally, Figure 6.2d shows how to create an ABAC policy rule to deny the browser (UID 10036) to access Google's homepage.

6.6 MOSES Evaluation

In this section, we report on the thorough experiments we run to evaluate the performance of MOSES. For all the experiments, we used a Google Nexus S phone.

6.6.1 Energy overhead

To measure the energy overhead produced by MOSES, we performed the following tests. We charged the battery of our device to the 100%. Then, every 10 minutes we run four system applications (sequentially) via a monkeyrunner [27] script: Calculator, Browser, Contacts and Email. For each of them the script performed common operations representative for the applications (multiplication of numbers in case of Calculator, browsing several webpages in case of Browser, calling a number and creating an account in case of Contacts, and composing and sending a email in case of Email application). Each experiment lasted for a total of 120 minutes. We executed this experiment for three types of systems: Stock Android, MOSES without *SP* changes, and MOSES with *SP* changes (the system switched between two profiles every 20 minutes).

During each experiment, every 10 seconds, our service measured the level of the battery and wrote this value into a log file. For each of the three considered systems, we executed the test 10 times and averaged the obtained values. The results of this experiment are reported in Figure 6.3. We note that the curves for the three considered systems behave similarly. This shows that the fact that MOSES is just running, or even switching between context does not incur a noticeable energy overhead. Since the goal of the experiment was

[1] The demo presented at [124] is available on our website [28].

only to evaluate the overhead compared to Taintdroid and Stock Android, rather than to calculate the absolute energy consumption of MOSES, we consider this approach as sufficient for the purpose to approximate energy usage.

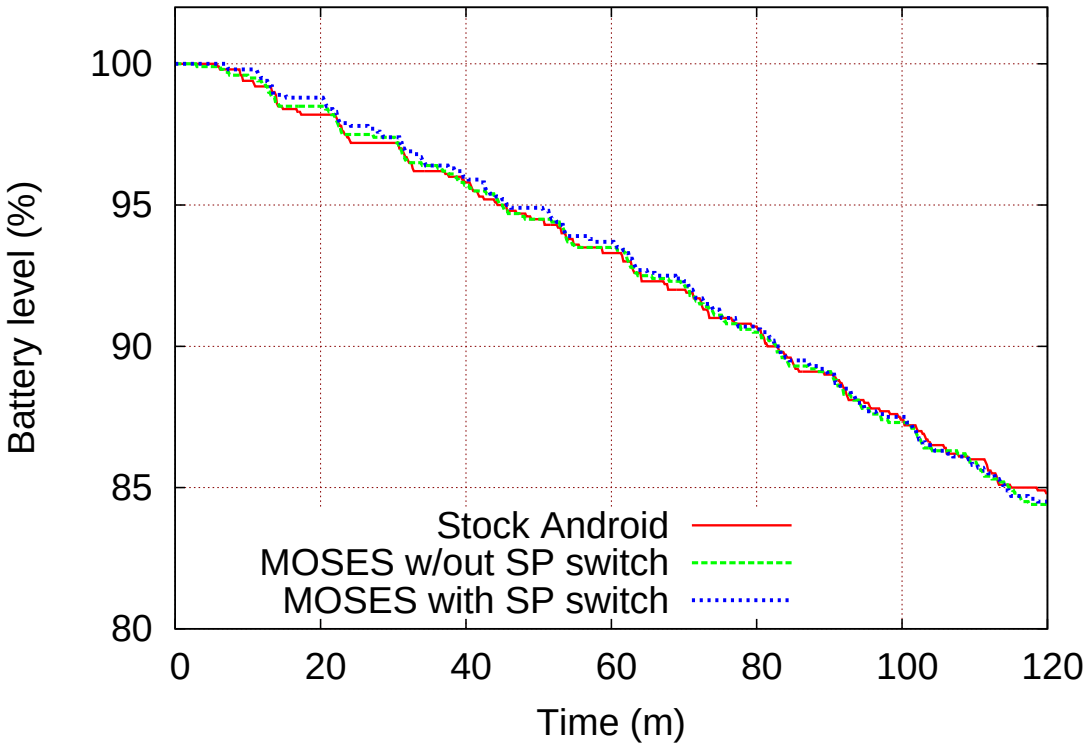


Figure 6.3: Energy overhead

6.6.2 Storage overhead

One of the most significant overheads produced by MOSES is the storage overhead. In fact, the separation of data for different SPs means that some application information will be duplicated in different profiles.

In general, the storage size consumed by a system can be expressed by the following equation:

$$size = size(OS) + \sum_{j=1}^{k} size(AE_j) + \sum_{j=1}^{k} size(AD_j), \tag{6.1}$$

where OS is the operating system, AE_j and AD_j, are the application executables and the application data or the j^{th} application. In the specific case of MOSES, $size(AD)$ is equal to:

$$size(AD_{MOSES}) = \sum_{i=1}^{n+1} \sum_{j=1}^{k} (size(AD_{ij})), \tag{6.2}$$

where $size(AD_{ij})$ is the size of the data of the j^{th} application in the i^{th} *SP*, k is the number of installed applications, and n is the number of *SPs*. One additional copy of application data (i.e., the $(n+1)$-th one) is required to store initial information of all applications. If a new *SP* is created, we need a "clean" copy of application data to be replicated into this new profile. Hence, MOSES stores a copy of application data just after the installation of the application: this copy is later used for replication when a new *SP* is created. It should be mentioned that for MOSES only the initial data of applications are duplicated. The data produced by applications during runtime are not replicated between *SPs*. Secondly, the data of applications, which are not allowed in a profile, are not copied into the profile.

When comparing MOSES with competitor approaches, MOSES produces less storage overhead. For instance, in case of mobile virtualization [26, 31, 48] not only application data are duplicated (as for MOSES), but also application executables and an operating system (sometimes partially [44]). Dual persona approaches [13, 16, 98, 141] additionally should have a separate copy of application executables in different profiles. Thus, MOSES adds less overhead comparing to this set of approaches because it only works with one copy of application executables.

Moreover, other improvements (currently left as future works) are possible for MOSES, e.g., currently for each *SPs* MOSES stores its own copy of the shared libraries of an application, instead they could be shared among the different profiles.

6.6.3 Microbenchmark

To assess the overall performance of our system, we decided to run a set of experiments with a benchmarking system. In particular, we used the Java microbenchmark CaffeineMark (version 3.0) [14] ported on the Android platform. This benchmark runs a set of tests which allows a user to assess different aspects of virtual machine performance. The benchmark does not produce absolute values for the tests. Instead it uses internal scoring measures, which are useful only in case of comparison with other systems. The overall score of CaffeineMark 3.0 is a geometric mean of all individual tests. That is why, to assess MOSES we decided to compare it with Stock Android system (version 2.3.4_r1) and Taintdroid system [70] (based on the Android system version 2.3.4_r1). We included in the comparison Taintdroid because MOSES incorporates its functionality, so we wanted to highlight the additional overhead that MOSES introduces compared to Taintdroid. We ran each benchmark 10 times.

For CaffeineMark 3.0 Java benchmark the observed results are reported in Figure 6.4.

From the figure, we notice that the results for Taintdroid and MOSES are almost the same: the checks that MOSES implements on top of Taintdroid do not have a significant influence on the results of this benchmark. Meanwhile, the difference of overall scores between unmodified (Stock Android) and modified systems (either Taintdroid or MOSES) is quite big. In fact, we can observe the performances are reduced by a 34%: the benckmark scores are 5910.7 for Stock Android, while 3895.9 for Taintdroid and 3923.3 for MOSES, the main contributors to this overhead are Loop (about 51% overhead) and Float (48%) tests.

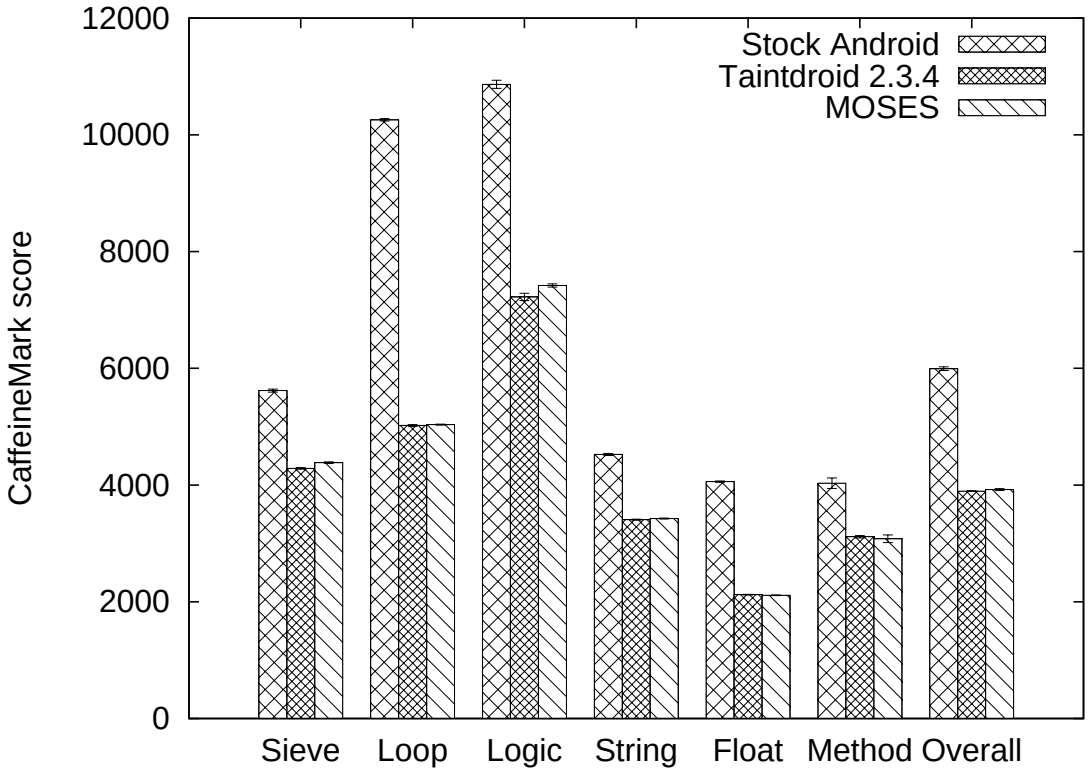


Figure 6.4: CaffeineMark Java benchmark results (with standard deviation)

6.6.4 Security Profile Switch Overhead

In this section, we present the results of the experiments measuring the time required to switch between *SPs*. We remind that during the profile switch (from an "old" to a "new" profile), MOSES performs the following operations: the unmounting of the data folders of the old profile, the mounting of data folders of the new profile, the unloading of the old and the loading of the new *Special Rules*. Therefore, the time to switch between *SPs* should depend on the number of *Special Rules* and the number of user apps. To find out the dependency between the time and these parameters we ran two sets of experiments. First, we measured the time required to switch *SPs* varying the number of user applications.

Then, we did the same measurement while changing the number of *Special Rules*. To measure this time, we put a call `SystemClock.elapsedRealtime()` before and after the switching operations, and calculated the difference between the values produced by this function.

To explore the dependency between the time and the number of applications we varied the number of user applications from 0 to 10. For each number of applications, a clean MOSES system was used (i.e. the system had been flashed on the phone just before the experiment). Then, a *SP* was created allowing all applications to be launched. Then, we measured the time of switch between this new profile and `DEFAULT` *SP*. For each number of applications we repeated the switch for 20 times and then calculated the average time of the switch. For all experiments the same set of 10 applications was used.

The results are shown in Figure 6.5a. From this figure, we observe that the switching time increases with the number of applications: moving from 1962 ms for 0 applications to 3496 ms for 10 applications. The rise of the time is associated with the increase of mounting and unmounting operations that MOSES performs during the switch (see Section 6.5.4). Furthermore, we note that the time is not uniformly rising with the growth of the number of applications. In particular, after the third application we observe a sharp increase of the function. The explanation of this phenomena is the following. The third application (named *com.antivirus*) after the installation starts a service that opens a file (`google_analytics.db`) and keeps it opened. Thus, MOSES has to kill the service before the unmounting could be performed successfully. In fact, MOSES system is designed in such a way that at first it simply tries to unmount the folder. Then, if this operation is unsuccessful, it sends to the blocking process a `SIGTERM` signal and tries to unmount again. If this try fails then MOSES kills the process, which holds a file opened, and performs the unmounting. Between the different tentatives, MOSES sleeps for 200 ms. We observe that the main time overhead is brought by these unsuccessful unmountings. The spread between 3 and 10 applications is merely about 250 ms.

The second experiment was conducted similarly to the first one, but in this case we varied the number of *Special Rules* assigned to a new *SP*: from 0 to 100, increasing by 10 rules each time. The results of this experiment are reported in Figure 6.5b. As we can see, the time of the switch slowly increases with the increase of the number of rules. We can also note that the standard deviation for the reported values is significant.

We also noticed that the time for the first change of profiles is considerably higher than for the following switches (although this cannot be inferred from the graphics which report average values). For instance, for 5 applications the time of the first switch is 9172 ms while for the second is just 3431 ms. In fact, during the first switch, for each application MOSES has to copy the initial data of an application to a new profile. That is why, the time for the first switch is several times higher than for the following switches. This fact also explains the wide spread of the standard deviation on the graphics.

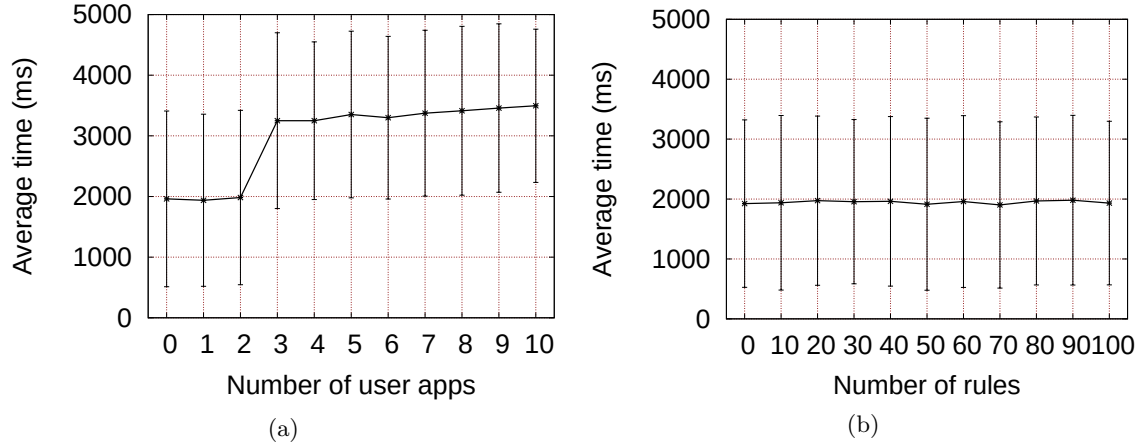


Figure 6.5: Time for profile switch (with standard deviation) as a function of the number of: (a) User applications, (b) Special Rules

6.6.5 Overheads of fine-grained control

As for MOSES fine-grained control overhead, it is mainly due to the checks of ABAC rules. To assess this overhead, we developed three different applications. The first application gets the IMEI of the phone, and stores it into the Android log. The second app reads the information about 10 contacts from the address book of a phone. The third one writes 1000 characters in a file. We measured the time of the checked operation using system function `SystemClock.elapsedRealtime()`. We ran these applications on different systems: Stock Android, Taintdroid (version 2.3.4) and on several variants of MOSES system, which differ from each other by the number of ABAC rules. In particular, we varied the number of rules for each app from 0 to 100, therefore, the total number of rules in the system changed from 0 to 300 (since there were three different applications). All ABAC rules for an app were the same. An example of ABAC rule for filesystem application (`MsFsTester`) is provided in Listing 6.2. During the assignment of the rules to *SP*, we assigned the same priority to all of them. We underline that this means considering the worst case, since during the check MOSES has to consider each rule. For each system, we ran each type of check for 1000 times and calculated the average, standard deviation and overhead for each variant of the systems. The results are summarized in Table 6.1. *Moses_010* in "System" column means that there are 10 ABAC rules for each considering operation resulting to total 30 rules entered into the system. Similarly, *Moses_000* means that there are no rules in MOSES system assigned to the current *SP*.

During the *SP* change `MosesRulesManager` selects from the database all ABAC rules related to the new profile and stores them in a special hashmap, where a key is equal to a tuple $(UID, operation)$ and the value corresponding to this key has the form $List < Map < (target, priority) >>$. Additionally, to each target an identifier of the ABAC rule is attached. Thus, at the first step the `MosesRulesManager` selects from the hashmap the

list of rules, which has been defined for an particular application (using UID) and for particular operation (e.g., write to a file). Then, during the second step `MosesRulesManager` iterates over the list and tries to find a match (using string comparison) between the target specified in the rule and the target provided by the system. If the match is found and the priority of this target is higher then the priority of the previous match (or -1 if it is the first time when match happens), then the system updates the current priority. When all items in the list are considered, the `MosesRulesManager` selects the identifier attached to the target with the highest priority that matches to the target provided by the system. Using this identifier `MosesRulesManager` selects the decision assigned to this ABAC rule. Thus, the time overhead during the check against the ABAC rules is emerged during these two steps.

Hashmap implementation provides constant time for getting an item, so the time overhead during the first step is not influenced considerably by the number of ABAC rules in the system. At the same time, the overhead during the second step depends on the number of rules. During this step, *MosesRulesManager* needs to iterate over the list of items and compare each target against the target attribute (we also support partial match). Our test cases have been designed assuming the worst case, i.e., for each operation all the list will be iterated and every target will be checked. In real-world scenarios, we assume that the number of entries in the list for each tuple $(UID, operation)$ will be smaller so the overhead will be smaller.

```
1 "UID of MsFsTester" "Write to a File" "/data/data/org.mosesdroid.msfstester/files/test"
2 "ALLOW" with scope "WORK"
```

Listing 6.2: Example of ABAC rule

Table 6.1: Operation time: average (AV), standard deviation (SD), overhead (OV)

SYSTEM	Operations								
	Get Device ID			Query ContentProvider			Write to a File		
	AV, ms	SD, ms	OV, %	AV, ms	SD, ms	OV, %	AV, ms	SD, ms	OV, %
Stock Android	1.018	0.700	0.00	10.516	7.653	0.00	0.818	2.893	0.00
Taintdroid	1.190	0.775	16.90	12.768	8.140	21.41	0.821	2.921	0.37
Moses_000	1.854	0.896	82.12	20.228	8.682	92.35	1.890	2.760	131.05
Moses_010	1.948	0.954	91.36	20.444	9.155	94.41	2.154	2.808	163.33
Moses_020	1.951	0.911	91.65	20.721	9.419	97.04	2.190	2.777	167.73
Moses_030	2.003	1.876	96.76	20.820	8.849	97.98	2.240	2.891	173.84
Moses_040	2.020	0.905	98.43	20.879	9.362	98.55	2.368	2.786	189.49
Moses_050	2.018	0.849	98.23	20.772	9.039	97.53	2.459	2.856	200.61
Moses_060	2.018	0.907	98.23	21.018	9.188	99.87	2.459	2.624	200.61
Moses_070	2.027	1.094	99.12	20.894	8.792	98.69	2.586	3.249	216.14
Moses_080	2.036	0.893	100.00	21.195	9.323	101.55	2.619	2.716	220.17
Moses_090	2.117	0.775	107.96	21.405	9.483	103.55	2.640	2.661	222.74
Moses_100	2.127	0.873	108.94	21.096	8.937	100.61	2.654	2.526	224.45

The developed applications represent three different check strategies that are implemented in MOSES. As described in Section 6.5.4, for each operation which MOSES can check, there is a separate class implemented in the `framework` library. According to the first strategy, the hook for MOSES check is embedded into the `framework` library. In

this case, the hook simply calls the check method of the corresponding operation class and enforces the result of the check. For instance, this strategy is used for "Get Device ID" operation check. In the second strategy, the hook is also placed into the `framework` library, but in the corresponding operation classes the additional checks against *ContentProvider*'s shadow database are performed. A representative of this strategy is the check of "Query ContentProvider" operation. In the third strategy, the hook is placed into the `core` library while the check is performed in the `framework` library. The call of the check method in the corresponding operation class is performed using Java reflection. The strategy is used for "Write to a File" operation check. Thus, *MsImeiTester*, *MsCpTester* and *MsFsTester* were developed to assess the overheads of these three strategies correspondingly.

Comparing the results obtained for Stock Android, Taintdroid and Moses_000, we can see that the main time overheads are added by MOSES operation checks. In fact, Taintdroid adds 16.9% time overhead in case of "Get Device ID" operation, 21.4% for "Query ContentProvider" and 0.4% in case of "Write to a File" operation, while MOSES even with no ABAC rules adds 82.1%, 92.4% and 131.1% overheads correspondingly comparing with Stock Android. Further, the results show that the most time consuming operation check is "Write to a File". Not surprisingly, because Java reflection used in the third MOSES check strategy is a quite expensive operation.

As we expected, time overheads grow if we increase the number of rules. We can see that Moses_000 adds 82.1%, 92.4% and 131.1% overheads. At the same time, Moses_100 adds 108.9%, 100.6% and 224.5%. We can see that relative overhead for "Get Device ID" operation is higher than for "Query ContentProvider". On the other hand, the absolute overhead is 0.273 ms for the first operation and 0.868 ms for the second operation. Thus, the small percentage in case of the second operation can be explained simply by the fact that it takes more time to process the results of the operation in the test application.

The absolute values of overheads of our three operations between Moses_000 and Moses_100 are 0.273 ms, 0.868 ms and 0.764 ms, respectively. The difference between the values is connected with the fact that it takes different time to process *target* attribute type in case of fixed and volatile *targets*. In case of fixed *target*, there is no need to compare the attribute value with the pattern. On the contrary, in case of volatile *target* MOSES has to compare the *target* value with the pattern in each relevant ABAC rule.

We assume that in a production system the total number of rules for a *SP* may be higher then the maximum number considered in our experiments (because the number of applications in a production system might be higher). At the same time, the number of rules for a tuple $(UID, operation)$ is smaller in real-world scenarios than considered in our experiments. So as the tuple $(UID, operation)$ serves as an index in our system, the number of rule checks in a production system, which causes the main part of the time overhead, will be smaller than in the considered number during the experiments.

Therefore, we assume that the main part of time overhead in a real-world system will be caused not by the number of ABAC rules in the system but by the embedded MOSES check itself. Unfortunately, this is the price we have to pay providing additional security mechanisms.

Chapter 7

Conclusion

During the last several years the popularity of mobile phones has increased greatly, and not the least role in the success of smartphones plays their ability to run third-party applications. Being constantly carried around with their users, equipped with lots of sensors and communicating with core user services as email, smartphones have become a very valuable source of information about their holders. Yet, these data are of particular interest not only to the legitimate owners of the devices. Mobile ad frameworks profile the behavior of users (e.g., the places they visit) to provide targeted and customized ads; and developers of third-party applications also embed the data-collection functionality [70, 84]. Moreover, there are also adversaries who develop their applications with malicious purposes, i.e., to collect as much as possible information about device owners and to perform malicious actions, e.g., send premium SMSs or turn the device into a bot. Not surprisingly, in this situation the users have a strong motivation to safeguard their devices from being misused and want to protect their privacy.

Understanding the fact that security is not a state but a process, in this thesis we propose our improvements to security of mobile ecosystems, primarily Android. In this chapter we summarize the key contributions done during the Ph.D. study and provide the possible directions of the future work.

7.1 Dissertation Summary and Future Work

The contribution of this dissertation consists of four main parts: 1) we proposed an approach that can detect repackaged Android applications in a fast way; 2) we developed a static-dynamic analyzer of Android applications managed to work in case of dynamic code updates; 3) we introduced an architecture of an attestation service for the Android platform [147]; 4) we developed a lightweight system that supports security profiles for the Android operating system [124, 150]. In addition to these contributions, we extensively described the security model of the Android operating system.

7.1.1 Fast Detection of Repackaged Android Applications

In this part of the thesis we presented an approach able to detect Android application repackaging based on the apk resource files. We implemented our approach in a tool called FSquaDRA. Leveraging hash files of resources already present in apks, FSquaDRA is capable of fast pairwise apk comparison. It computes the Jaccard similarity score for compared apks and classifies them as similar if substantial number of resource files are the same in both packages.

We have evaluated practicality of FSquaDRA in two aspects: whether it gives results similar to the code-based app repackaging detection techniques, and whether it is fast enough to handle significant number of apks. Our results are encouraging. The FSquaDRA resource similarity score is strongly correlated with the AndroGuard code similarity score, especially for the apks signed with different certificates, and thus, potentially, plagiarized. FSquaDRA is also has good performance, as it was able to process a dataset of more than 55000 apks on a laptop in less than 80 hours. Notice that our implementation was not optimized for better performance, as it is single-threaded. Yet, the approach can be easily parallelized using different parallelization algorithms for pairwise comparison.

The obvious limitation of the current tool is that an adversary who is familiar with the approach can easily change all resource files in the package to make his plagiarized application virtually undetectable by FSquaDRA. Resource similarity metrics can be hardened against this by looking into files themselves rather than just comparing the digests, but it will lead to performance losses (which can become comparable with those of the code-based repackaging detection techniques if implemented reasonably). The most promising, to our point of view, is a hybrid approach, when repackaged applications are detected using both approaches, code and resource comparison. We believe this is a very interesting research direction.

Another interesting direction is to look into the data produced by FSquaDRA searching for patterns and interesting findings, such as the fact that on average applications signed with the same certificate have higher code similarity score than resource similarity score, while this difference is not so evident in the apps signed with different certificates.

FSquaDRA opens an avenue of enhancement for app plagiarism detection algorithms, and not only for Android. For other ecosystems, such as iOS or Windows Phone, that request the developers to submit the full source code and resources before publishing apps on the market our technique can be used to improve the on-market plagiarism detection algorithms by complementing the code similarity-based approaches.

7.1.2 Static-Dynamic Analyser of Android Apps in the Presence of Reflection and Dynamic Class Loading

Modern applications make an extensive use on the dynamic capabilities, namely reflection and dynamic class loading, available in Android OS. Being adopted from Java, these techniques in Android incur an additional threat because the loaded code receives the same privileges as the loading one. Malware apps can then leverage on these capabilities to conceal their malicious behavior from analyzers, while looking normal applications.

To address Issue 3 we presented StaDynA, a technique that interleaves static and dynamic types of analysis in order to scrutinize application code in the presence of reflection and dynamic class loading, and an implementation of this technique on the Android platform. Our approach makes it possible to compute the method call graph of an application and to expand it by capturing additional modules loaded at runtime and additional paths of execution obfuscated by reflection calls. In order to produce the expanded call graph StaDynA does not require the modification of the application itself. The results produced by StaDynA can then be fed to state of art analyzers in order to improve their precision (for instance, for reachability analysis over the obtained MCG). Thus, StaDynA may help malware analysts by increasing they ability to detect suspicious behavior.

However, our tool has space for future improvements. For StaDynA the coverage (the percentage of triggered MOIs) is especially important. Currently our system uses a semi-automatic approach. This means that a user (or the system) should trigger the methods, which contain DCL and reflection calls. We evaluated StaDynA manually, and we were not able to trigger all MOIs in the apps because our triggering is mostly GUI-based (system events are also processed by StaDynA but it is difficult if not impossible for a human analyst to produce a sufficient range of system events that might trigger a MOI). Notice that only the triggering limits the coverage of MOIs.

As a way to improve StaDynA we plan to implement a fully automatic approach for triggering. The first research proposals in this direction automate the user behavior [83, 89, 120, 151], and they might be integrated with StaDynA. However, the system events triggering still needs to be explored further.

Another possible direction to reduce the amount of manual work is to resolve the targets of reflection calls statically (e.g., see [106]). The analysis performed in [74] shows that it is possible to resolve automatically the targets of reflection calls in 59% of applications that use reflection. At the same time, the analysis was performed for the "close world" scenario, which is not realistic today, given that dynamic class loading is a popular technique for modern apps. Additionally, currently reflection is used more heavily. While the article [74] reports that only 61% of apps use reflection, our analysis performed on a similar dataset shows that nowadays about 88% of applications use this technique. Additional benefit of resolving reflection call targets at static time is the increase of code coverage.

Usually, dynamic analysis allows an expert to explore only one execution path at a time. However, dynamic traces may differ depending on the context of the execution, e.g., some methods may contain calls invoked with parameters, according to which the reflection call target may change. Therefore, another possible direction of improving StaDynA is to incorporate information obtained during different runs of analysis.

StaDynA also has other limitations. Its analysis is based on the UID of an application. However, it is possible in Android that several apps have the same UID. In this case, StaDynA will also collect the information produced by other apps with the same UID. At the same time, this information will not be used to complement MCG, and the calls will be considered by StaDynA as suspicious. Currently, StaDynA is also unable to download and analyze the code that is loaded directly into the memory [128]. However, the hacks considered in [128] are not "officially" recommended to use because they avoid the code optimization phase performed by Android.

7.1.3 Attestation Service for the Android Platform

The openness of Android has provided possibilities for other players (e.g., Amazon) to run their own markets of applications. At the same time, the popularity of the platform also attracts adversaries. The last reports show that even the users of the official Google Play market cannot feel safe because of the lack of the strict vetting process there.

To address this issue we introduced TruStore (Trusted Store) to enhance app trust. The main challenge for enabling a trusted store for Android is preservation of openness of the ecosystem. Our solution is intended to build a secure infrastructure for provisioning validated apps and their updates. We have designed and developed the client part of the TruStore architecture, showing that implemented modifications to the Android platform can enable this functionality. TruStore does not hinder the app management process for end-users and developers, but it can provide the app trustworthiness assurance for end-users and enterprises enabling the BYOD programs.

7.1.4 Supporting and Enforcing Security Profiles

To address Issue 1 we developed MOSES, a solution for separating modes of use in smartphones. MOSES implements soft virtualization through controlled software isolation. MOSES is the first solution to provide policy-based security containers implemented completely via software. By acting at the system level we prevent applications to be able to bypass our isolation. Basically, MOSES provides a possibility to separate different aspects of user life activity specifying several *Security Profiles* on a smartphone. Such solution provides a user to use the same device in different environemts, e.g., work and personal, providing the control of the profiles to the interested parties.

With MOSES each security profile can be associated to one or more contexts that

determine when the profile become active. Contexts are defined in term of low level features (e.g., time and location) and high level features (reputation, trust level, etc.). Switching between *Security Profiles* can require users interaction or be automatic, efficient, and transparent to the user basing on defined contexts. Both contexts and profiles can be easily and dynamically specified by end users. For this purpose MOSES provides a special GUI application. Profiles can be fine-grained to the level of single object (e.g., file, SMS) and single application.

We implemented MOSES and run a thorough set of experiments to evaluate its efficiency and effectiveness. The experiments show the feasibility and accepted performance of our solution for storage and energy consumption.

However, at the present moment MOSES has also some limitations. At first, fine-grained policies and allowed applications are specified using the UID of an application. Meanwhile, in Android it is possible that some applications share the same UID. Thus, if we apply MOSES rules and restrictions to one application they automatically will be extended to the other ones with same UID. Furthermore, some fine-grained policies in MOSES are built on top of Taintdroid [70] functionality. Thus, MOSES inherits the limitations of Taintdroid explained in Section 6.2. It should be also mentioned that the applications that have root access to the system can bypass MOSES protection. Thus, MOSES is ineffective in combating with the malware that obtains root access, e.g., rootkits.

MOSES can also be improved in several aspects. For instance, to make the policy specification process easier, a solution could be to embed into the system policy templates that can be simply selected and associated to an application. It should be also mentioned that currently MOSES does not separate system data (e.g., system configuration files) and information on SD cards. In the future we plan to add this functionality to the system. Moreover, performance overheads are also planned to be reduced considerably in the future versions.

Bibliography

[1] ActionBarSherlock. Available Online. http://actionbarsherlock.com/.

[2] AndroGuard: Reverse engineering, Malware and goodware analysis of Android applications. Available Online. https://code.google.com/p/androguard/.

[3] Android-apktool: A tool for reverse engineering Android apk files. Available Online. https://code.google.com/p/android-apktool/.

[4] Android application: Building and Running. Available Online. http://developer.android.com/tools/building/index.html.

[5] Android Security Discussions: Multiple Certificates and Upgrade process. Available Online. https://groups.google.com/forum/?fromgroups#!topic/android-security-discuss/sY7Ormv3uWk.

[6] Android Security Overview. Available Online. http://source.android.com/devices/tech/security/index.html.

[7] AndroidBest – Android market. http://androidbest.ru/.

[8] AndroidDrawer – Android market. http://www.androiddrawer.com/.

[9] AndroidLife – Android market. http://androidlife.ru/.

[10] Anruan – Android market. http://www.anruan.com/.

[11] AppsApk – Android market. http://www.appsapk.com/.

[12] Are Your Sales Reps Missing Important Sales Opportunities? Available Online. http://http://m.sybase.com/files/White_Papers/Solutions_SAP_Reps.pdf/.

[13] AT&T Toggle. Available Online. https://www.wireless.att.com//businesscenter/solutions/industry-solutions/mobile-productivity-solutions/toggle.jsp.

[14] CaffeineMark 3.0 Benchmark. Available Online. http://www.benchmarkhq.ru/cm30/.

[15] capabilities(7) - Linux man page. Available Online. `http://linux.die.net/man/7/capabilities`.

[16] Divide webpage. Available Online. `http://www.divide.com/`.

[17] Dual Persona Definition. Available Online. `http://searchconsumerization.techtarget.com/definition/Dual-persona`.

[18] F-Droid – Android market. `https://f-droid.org/`.

[19] Fixmo SafeZone: Corporate Data Protection. Available Online. `http://fixmo.com/products/safezone`.

[20] Gartner Says Smartphone Sales Accounted for 55 Percent of Overall Mobile Phone Sales in Third Quarter of 2013. Available Online. `http://www.gartner.com/newsroom/id/2623415`.

[21] Good BYOD solutions. Available Online. `http://www1.good.com/mobility-management-solutions/bring-your-own-device`.

[22] Google Play – Android official market. `https://play.google.com/store/apps`.

[23] Jar File Specification. Available Online. `http://docs.oracle.com/javase/6/docs/technotes/guides/jar/jar.html`.

[24] jarsigner - JAR Signing and Verification Tool. Available Online. `http://docs.oracle.com/javase/6/docs/technotes/tools/windows/jarsigner.html`.

[25] Jdk 1.1 new feature summary. Available Online. `http://www.tns.lcs.mit.edu/manuals/java-1.1.1/relnotes/features.html`.

[26] Mobile Virtualization. Available Online. `http://www.redbend.com/en/products-services/mobile-virtualization`.

[27] monkeyrunner. Available Online. `http://developer.android.com/tools/help/monkeyrunner_concepts.html`.

[28] MOSES project. Available Online. `http://mosesdroid.org/`.

[29] mount(2) - Linux man page. Available Online. `http://linux.die.net/man/2/mount`.

[30] NitroDesk TouchDown. Available Online. `http://www.nitrodesk.com/TouchDown.aspx`.

[31] OKL4 Microvisor. Available Online. `http://www.ok-labs.com/products/okl4-microvisor`.

[32] PandaApp – Android market. `http://android.pandaapp.com/`.

[33] Permissions for System Apps (not in /data/system/packages.xml?). Forum Discussion. `https://groups.google.com/forum/#!topic/android-developers/ZOrtSBG5_XA`.

[34] Security Tips. Available Online. `http://developer.android.com/training/articles/security-tips.html`.

[35] SlideME – Android market. `http://slideme.org/`.

[36] Smali: An assembler/disassembler for Android's dex format. Available Online. `https://code.google.com/p/smali/`.

[37] System Permissions. Available Online. `http://developer.android.com/guide/topics/security/permissions.html`.

[38] The Android Open Source Project. Available Online. `http://source.android.com/index.html`.

[39] Ubuntu Manuals - PAM namespaces. Available Online. `http://manpages.ubuntu.com/manpages/maverick/man8/pam_namespace.8.html`.

[40] Unisys Establishes a Bring Your Own Device (BYOD) Policy. Available Online. `http://www.insecureaboutsecurity.com/2011/03/14/unisys_establishes_a_bring_your_own_device_byod_policy/`.

[41] Filesystem Hierarchy Standard, version 2.3. Available Online, January 2004. `http://www.pathname.com/fhs/pub/fhs-2.3.html`.

[42] Yuvraj Agarwal and Malcolm Hall. ProtectMyPrivacy: Detecting and Mitigating Privacy Leaks on iOS Devices Using Crowdsourcing. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 97–110, 2013.

[43] Musheer Ahmed and Mustaque Ahamad. Protecting Health Information on Mobile Devices. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 229–240, 2012.

[44] Jeremy Andrus, Christoffer Dall, Alexander Van't Hof, Oren Laadan, and Jason Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 173–187, 2011.

[45] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 217–228, 2012.

[46] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. AppGuard – Enforcing User Requirements on Android Apps. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, pages 543–548, 2013.

[47] Guangdong Bai, Liang Gu, Tao Feng, Yao Guo, and Xiangqun Chen. Context-Aware Usage Control for Android. In *Proceedings of the Conference on Security and Privacy in Communication Networks*, SecureComm, pages 326–343, 2010.

[48] Ken Barr, Prashanth Bungale, Stephen Deasy, Viktor Gyuris, Perry Hung, Craig Newell, Harvey Tuch, and Bruno Zoppis. The VMware Mobile Virtualization Platform: Is That a Hypervisor in Your Pocket? *SIGOPS Oper. Syst. Rev.*, 44(4):124–135, December 2010.

[49] Alastair R Beresford, Andrew Rice, and Nicholas Skehin. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*, HotMobile '11, pages 49–54, 2011.

[50] Daniel G. Bobrow, Richard P. Gabriel, and Jon L. White. Object-oriented programming. chapter CLOS in Context: The Shape of the Design Space, pages 29–61. MIT Press, 1993.

[51] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 241–250. ACM, 2011.

[52] Jeff Bogda and Ambuj Singh. Can a Shape Analysis Work at Run-time? In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM'01, pages 2–2, 2001.

[53] Joseph Bradley, Jeff Loucks, James Macaulay, Richard Medcalf, and Lauren Buckalew. BYOD: A Global Perspective. Harnessing Employee-Led Innovation. Available Online, 2012. `http://www.cisco.com/web/about/ac79/docs/re/BYOD_Horizons-Global.pdf`.

[54] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, NDSS '12, 2012.

[55] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. Practical and Lightweight Domain Isolation on

Android. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 51–62, 2011.

[56] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 131–146, 2013.

[57] Hao Chen, David Wagner, and Drew Dean. Setuid Demystified. In *Proceedings of the 11th USENIX Security Symposium*, pages 171–190, 2002.

[58] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in Android. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 239–252, 2011.

[59] Fred Chung. Custom Class Loading in Dalvik. Available Online. `http://android-developers.blogspot.it/2011/07/custom-class-loading-in-dalvik.html`.

[60] Rudi Cilibrasi and Paul M. B. Vitányi. Clustering by compression. *IEEE Transactions on Information Theory*, 51:1523–1545, 2005.

[61] Christian Collberg, Ginger Myles, and Andrew Huntwork. Sandmark–A Tool for Software Protection Research. *IEEE Security and Privacy*, 1(4):40–49, July 2003.

[62] Mauro Conti, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich. CRêPE: A system for enforcing fine-grained context-related policies on Android. *IEEE Transactions on Information Forensics and Security*, 7(5):1426–1438, 2012.

[63] Jonathan Crussell, Clint Gibler, and Hao Chen. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *Proceedings of the 17th European Symposium on Research in Computer Security*, ESORICS '12, pages 37–54, 2012.

[64] Jonathan Crussell, Clint Gibler, and Hao Chen. Scalable SEmantics-Based Detection of Similar Android Applications. In *Proceedings of the 18th European Symposium on Research in Computer Security*, ESORICS '13, 2013.

[65] Christoffer Dall and Jason Nieh. KVM for ARM. In *Proceedings of the 12th Annual Linux Symposium*, 2010.

[66] Anthony Desnos. Android: Static Analysis Using Similarity Distance. In *Proceedings of the 2012 45th Hawaii International Conference on System Sciences*, HICSS '12, pages 5394–5403, 2012.

[67] Technische Universitat Dresden and University of Technology Berlin. L4Android. Available Online. `http://l4android.org/`.

[68] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. An empirical study of cryptographic misuse in android applications. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 73–84, 2013.

[69] Manuel Egele, Christopher Kruegel, Engin Kirda, and Giovanni Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the 18th Annual Network & Distributed System Security Symposium*, NDSS '11, 2011.

[70] 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*, OSDI'10, pages 1–6, 2010.

[71] William Enck, Damien Octeau, Patrick McDaniel, and Swarat Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Conference on Security*, SEC'11, pages 21–21, 2011.

[72] William Enck, Machigar Ongtang, and Patrick McDaniel. Understanding Android Security. *IEEE Security and Privacy*, 7(1):50–57, 2009.

[73] F-Secure. Trojan:Android/FakeNotify Gets Updated. Available Online, Dec. 2011. `http://www.f-secure.com/weblog/archives/00002291.html?tduid=f57e2769518f081721ffca586e797b2a`.

[74] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 627–638, 2011.

[75] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android Permissions: User Attention, Comprehension, and Behavior. In *Proceedings of the Eighth Symposium on Usable Privacy and Security*, SOUPS '12, pages 3:1–3:14, 2012.

[76] Denis Feth and Christian Jung. Context-Aware, Data-Driven Policy Enforcement for Smart Mobile Devices in Business Environments. In *Proceedings of Security and Privacy in Mobile Information and Communication Systems*, MobiSec '12, pages 69–80, 2012.

[77] Denis Feth and Alexander Pretschner. Flexible Data-Driven Security for Android. In *Proceedings of the 2012 IEEE Sixth International Conference on Software Security and Reliability*, SERE '12, pages 41 –50, 2012.

[78] E. Fragkaki, L. Bauer, L. Jia, and D. Swasey. Modeling and Enhancing Androids Permission System. In *Proceedings of the 17th European Symposium on Research in Computer Security (ESORICS '13)*, volume 7459 of *Lecture Notes in Computer Science*, pages 1–18, 2012.

[79] Olga Gadyatskaya, Fabio Massacci, and Yury Zhauniarovich. Security in the Firefox OS and Tizen Mobile Platforms. *IEEE Computer*, (Special Issue on Mobile Application Security), 2014. to appear.

[80] Aleksandar Gargenta. Deep Dive into Android IPC/Binder Framework. Available Online. `https://thenewcircle.com/s/post/1340/Deep_Dive_Into_Binder_Presentation.htm`.

[81] Marko Gargenta. Android Security Underpinnings. Available Online. `https://thenewcircle.com/s/post/1518/Android_Security_Underpinnings.htm`.

[82] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural Detection of Android Malware Using Embedded Call Graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security*, AISec '13, pages 45–54, 2013.

[83] Andrea Gianazza, Federico Maggi, Aristide Fattori, Lorenzo Cavallaro, and Stefano Zanero. PuppetDroid: A User-Centric UI Exerciser for Automatic Dynamic Analysis of Similar Android Applications. Technical report, 2014.

[84] 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*, TRUST'12, pages 291–307, 2012.

[85] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. AdRob: examining the landscape and impact of Android application plagiarism. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 431–444, 2013.

[86] Peter Gilbert, Byung-Gon Chun, Landon P. Cox, and Jaeyeon Jung. Vision: Automated Security Validation of Mobile Apps at App Markets. In *Proceedings of the Second International Workshop on Mobile Cloud Computing and Services*, MCS '11, pages 21–26, 2011.

[87] Akhilesh Gupta, Anupam Joshi, and Gopal Pingali. Enforcing Security Policies in Mobile Devices Using Multiple Personas. In *MobiQuitous*, volume 73 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 297–302, 2010.

[88] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A Scalable System for Detecting Code Reuse Among Android Applications. In *Proceedings of the 9th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'12, pages 62–81, 2013.

[89] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. PUMA: Programmable UI-Automation for Large-Scale Dynamic Analysis of Mobile Apps. In *Proceedings of the 12th International Conference on Mobile Systems, Applications, and Services (MobiSys'14)*, 2014. to appear.

[90] Gernot Heiser. Virtualization for Embedded Systems. Technical report, Open Kernel Labs, Inc., 2007. `http://www.ok-labs.com/_assets/image_library/virtualization-for-embedded-systems1983.pdf`.

[91] Martin Hirzel, Amer Diwan, and Michael Hind. Pointer Analysis in the Presence of Dynamic Class Loading. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 3086 of *LNCS*, pages 96–122, 2004.

[92] Martin Hirzel, Daniel von Dinklage, Amer Diwan, and Michael Hind. Fast Online Pointer Analysis. *ACM Tran. on Programming Languages and Systems*, 29(2), 2007.

[93] Peter Hornyack, Seungyeop Han, Jaeyeon Jung, Stuart Schechter, and David Wetherall. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, pages 639–652, 2011.

[94] Cuixiong Hu and Iulian Neamtiu. Automating GUI Testing for Android Applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, 2011.

[95] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, 2009.

[96] Heqing Huang, Sencun Zhu, Peng Liu, and Dinghao Wu. A Framework for Evaluating Mobile App Repackaging Detection Algorithms. In *Proceedings of the 6th International Conference on Trust and Trustworthy Computing*, TRUST '13, pages 169–186, 2013.

[97] Joo-Young Hwang, Sang-Bum Suh, Sung-Kwan Heo, Chan-Ju Park, Jae-Min Ryu, Seong-Yeol Park, and Chul-Ryun Kim. Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones. In *Proceedings of the 5th Consumer Communications and Newtork Conference*, CCNC '08, pages 257 –261, 2008.

[98] Jinseong Jeon, Kristopher K. Micinski, Jeffrey A. Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S. Foster, and Todd Millstein. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 3–14, 2012.

[99] Xuxian Jiang. Security Alert: New Stealthy Android Spyware – Plankton – Found in Official Android Market. Available Online, June 2011. `http://www.csc.ncsu.edu/faculty/jiang/Plankton/`.

[100] Mona Erfani Joorabchi and Ali Mesbah. Reverse Engineering iOS Mobile Applications. In *Proceedings of the 2012 19th Working Conference on Reverse Engineering*, WCRE '12, pages 177–186, 2012.

[101] Alexandros Kapravelos, Yan Shoshitaishvili, Marco Cova, Christopher Kruegel, and Giovanni Vigna. Revolver: An Automated Approach to the Detection of Evasiveweb-based Malware. In *Proceedings of the 22nd USENIX Conference on Security*, SEC'13, pages 637–652, 2013.

[102] Palanivel Balaji Kodeswaran, Vikrant Nandakumar, Shalini Kapoor, Pavan Kamaraju, Anupam Joshi, and Sougata Mukherjea. Securing Enterprise Data on Smartphones Using Run Time Information Flow Control. In *Proceedings of the 13th IEEE International Conference on Mobile Data Management*, MDM '12, pages 300–305, 2012.

[103] Dean Kramer, Anna Kocurova, Samia Oussena, Tony Clark, and Peter Komisarczuk. An Extensible, Self Contained, Layered Approach to Context Acquisition. In *Proceedings of the Third International Workshop on Middleware for Pervasive Mobile and Embedded Computing*, M-MPAC '11, pages 6:1–6:7, 2011.

[104] Matthias Lange, Steffen Liebergeld, Adam Lackorzynski, Alexander Warg, and Michael Peter. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, pages 39–50, 2011.

[105] Sheng Liang and Gilad Bracha. Dynamic Class Loading in the Java Virtual Machine. In *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented*

Programming, Systems, Languages, and Applications, OOPSLA '98, pages 36–44, 1998.

[106] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection Analysis for Java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 139–160, 2005.

[107] Hiroshi Lockheimer. Android and Security. `http://googlemobile.blogspot.com/2012/02/android-and-security.html`, February 2012. Available Online.

[108] Alex Lockwood. Binders & Window Tokens. Available Online, July 2013. `http://www.androiddesignpatterns.com/2013/07/binders-window-tokens.html`.

[109] Denis Maslennikov. IT Threat Evolution: Q1 2013. Available Online, May 2013. `http://www.securelist.com/en/analysis/204792292/IT_Threat_Evolution_Q1_2013`.

[110] Roberto Mijat and Andy Nightingale. Virtualization is Coming to a Platform Near You. Technical report, ARM, 2010. `http://www.arm.com/files/pdf/System-MMU-Whitepaper-v8.0.pdf`.

[111] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, pages 328–332, 2010.

[112] John Oberheide. Dissecting the Android Bouncer. Available Online, 2012. `http://jon.oberheide.org/files/summercon12-bouncer.pdf`.

[113] Machigar Ongtang, Stephen McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 340–349, 2009.

[114] Pandalabs. New Malware Attack through Google Play. Available Online, Feb. 2014. `http://pandalabs.pandasecurity.com/new-malware-attack-through-google-play/`.

[115] Ketan Parmar. In Depth: Android Package Manager and Package Installer. Available Online, October 2012. `http://www.kpbird.com/2012/10/in-depth-android-package-manager-and.html`.

[116] Nicole Perlroth and Nick Bilton. Mobile Apps Take Data Without Permission. Available Online, February 2012. `http://bits.blogs.nytimes.com/2012/02/15/google-and-mobile-apps-take-data-books-without-permission/?_r=0`.

[117] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications. In *Proceedings of the 21st Annual Network & Distributed System Security Symposium*, NDSS '14, 2014.

[118] Rahul Potharaju, Andrew Newell, Cristina Nita-Rotaru, and Xiangyu Zhang. Plagiarizing smartphone applications: Attack strategies and defense techniques. In *Proceedings of the 4th International Conference on Engineering Secure Software and Systems*, ESSoS'12, pages 106–120, 2012.

[119] Emil Protalinski. Warning: New Android malware tricks users with real Opera Mini. Available Online, 2012 July. `http://www.zdnet.com/warning-new-android-malware-tricks-users-with-real-opera-mini-7000001586/`.

[120] Vaibhav Rastogi, Yan Chen, and William Enck. AppsPlayground: Automatic Security Analysis of Smartphone Applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 209–220, 2013.

[121] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. DroidChameleon: Evaluating Android Anti-malware Against Transformation Attacks. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 329–334, 2013.

[122] Steven B. Roosa and Stephen Schultze. Trust Darknet: Control and Compromise in the Internet's Certificate Authority Model. *IEEE Transactions on Internet Computing*, 17(3):18–25, 2013.

[123] Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. MOSES: Supporting Operation Modes on Smartphones. In *Proceedings of the 17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 3–12, 2012.

[124] Giovanni Russello, Mauro Conti, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich. Demonstrating the Effectiveness of MOSES for Separation of Execution Modes. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 998–1000, 2012.

[125] Giovanni Russello, Bruno Crispo, Earlence Fernandes, and Yury Zhauniarovich. YAASE: Yet Another Android Security Extension. In *Proceedings of the Third IEEE International Conference on Privacy, Security, Risk and Trust and Third International Conference on Social Computing*, SocialCom/PASSAT, pages 1033–1040, 2011.

[126] Giovanni Russello, Arturo Blas Jimenez, Habib Naderi, and Wannes van der Mark. FireDroid: Hardening Security in Almost-stock Android. In *Proceedings of the 29th Annual Computer Security Applications Conference*, ACSAC '13, pages 319–328, 2013.

[127] Thorsten Schreiber. Android Binder. Android Interprocess Communication. Master's thesis, Ruhr University Bochum, 2011.

[128] Patrick Schulz. Code Protection in Android. Available Online, 2012. `http://net.cs.uni-bonn.de/fileadmin/user_upload/plohmann/2012-Schulz-Code_Protection_in_Android.pdf`.

[129] Asaf Shabtai, Yuval Fledel, Uri Kanonov, Yuval Elovici, Shlomi Dolev, and Chanan Glezer. Google Android: A Comprehensive Security Assessment. *IEEE Security and Privacy*, 8:35–44, 2010.

[130] Stephen Smalley and Robert Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, NDSS '13, 2013.

[131] Brian Cantwell Smith. *Procedural Reflection in Programming Languages*. PhD thesis, Massachusetts Institute of Technology, Laboratory for Computer Science, 1982. `http://publications.csail.mit.edu/lcs/pubs/pdf/MIT-LCS-TR-272.pdf`.

[132] Dennis Sosnoski. Java programming dynamics, Part 1: Java classes and class loading. Available Online. `http://www.ibm.com/developerworks/library/j-dyn0429/`.

[133] Symantec. Securing the Mobile App Market: How Code Signing Can Bolster Security for Mobile Applications. White paper, 2012.

[134] Enea Android team. The Android boot process from power on. `http://www.androidenea.com/2009/06/android-boot-process-from-power-on.html`, June 2009. Available Online.

[135] TrendMicro. Mobile App Reputation Service, 2011. `http://www.trendmicro.co.uk/media/ds/mobile-app-reputation-service-datasheet-en.pdf`.

[136] Bart van Wissen, Nicholas Palmer, Roelof Kemp, Thilo Kielmann, and Henri Bal. ContextDroid: an Expression-Based Context Framework for Android. In *Proceedings of PhoneSense 2010*, pages 1–5, 2010.

[137] Timothy Vidas and Nicolas Christin. Sweetening Android Lemon Markets: Measuring and Combating Malware in Application Marketplaces. In *Proceedings of the*

Third ACM Conference on Data and Application Security and Privacy, CODASPY '13, pages 197–208, 2013.

[138] Tielei Wang, Kangjie Lu, Long Lu, Simon Chung, and Wenke Lee. Jekyll on iOS: When Benign Apps Become Evil. In *Proceedings of the 22Nd USENIX Conference on Security*, SEC'13, pages 559–572, 2013.

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

[140] Erik Ramsgaard Wognsen and Henrik Søndberg Karlsen. Static Analysis of Dalvik Bytecode and Reflection in Android. Master's thesis, Aalborg University, 2012.

[141] Rubin Xu, Hassen Saïdi, and Ross Anderson. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 27–27, 2012.

[142] Yang Xu, Felix Bruns, Elizabeth Gonzalez, Shadi Traboulsi, Klaus Mott, and Attila Bilgic. Performance Evaluation of Para-virtualization on Modern Mobile Phone Platform. In *Proceedings of the International Conference on Computer, Electrical, and Systems Science, and Engineering*, ICCESSE '10, 2010.

[143] Karim Yaghmour. Extending Android HAL. Available Online, September 2012. `http://www.opersys.com/blog/extending-android-hal`.

[144] Karim Yaghmour. *Embedded Android*. O'Reilly Media, Inc., 2013.

[145] Lok Kwong Yan and Heng Yin. DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 29–29, 2012.

[146] Eric Yuan and Jin Tong. Attributed Based Access Control (ABAC) for Web Services. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '05, pages 561–569, 2005.

[147] Yury Zhauniarovich, Olga Gadyatskaya, and Bruno Crispo. DEMO: Enabling Trusted Stores for Android. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 1345–1348, 2013.

[148] Yury Zhauniarovich, Olga Gadyatskaya, and Bruno Crispo. TruStore: Implementing a Trusted Store for Android. Technical Report DISI-14-010, Department of Engineering and Computer Science, University of Trento, May 2014.

[149] Yury Zhauniarovich, Olga Gadyatskaya, Bruno Crispo, Francesco La Spina, and Ermanno Moser. FSquaDRA: Fast Detection of Repackaged Applications. In *Proceedings of the 28th Annual IFIP WG 11.3 Working Conference on Data and Applications Security and Privacy*, DBSec '14, pages 131–146, 2014. to appear.

[150] Yury Zhauniarovich, Giovanni Russello, Mauro Conti, Bruno Crispo, and Earlence Fernandes. MOSES: Supporting and Enforcing Security Profiles on Smartphones. *IEEE Transactions on Dependable and Secure Computing*, 11(3):211–223, May 2014.

[151] Cong Zheng, Shixiong Zhu, Shuaifu Dai, Guofei Gu, Xiaorui Gong, Xinhui Han, and Wei Zou. SmartDroid: An Automatic System for Revealing UI-based Trigger Conditions in Android Applications. In *Proceedings of the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '12, pages 93–104, 2012.

[152] Yibing Zhongyang, Zhi Xin, Bing Mao, and Li Xie. DroidAlarm: An All-sided Static Analysis Tool for Android Privilege-escalation Malware. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 353–358, 2013.

[153] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. Fast, Scalable Detection of "Piggybacked" Mobile Applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 185–196, 2013.

[154] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. Detecting Repackaged Smartphone Applications in Third-party Android Marketplaces. In *Proceedings of the Second ACM Conference on Data and Application Security and Privacy*, CODASPY '12, pages 317–326, 2012.

[155] Yajin Zhou and Xuxian Jiang. An Analysis of the AnserverBot Trojan. Available Online, September 2011. `http://www.csc.ncsu.edu/faculty/jiang/pubs/AnserverBot_Analysis.pdf`.

[156] Yajin Zhou and Xuxian Jiang. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 95–109, 2012.

[157] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the 19th Annual Network & Distributed System Security Symposium*, NDSS '12, 2012.

[158] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and V.W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the 4th International Conference on Trust and Trustworthy Computing*, TRUST'11, pages 93–107, 2011.